VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Notifikace, asynchronní vykonávání úloh ve webových aplikacích, dotazování, cachování a škálování technologií pro ukládání dát

# Notification, asynchronous execution of tasks in web applications, polling, caching and scaling of storage technologies

2019 Andrej Pinčík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Andrej Pinčík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Notifikace, asynchronní vykonávání úloh ve webových aplikacích, dotazování, cachování a škálování technologií pro ukládání dat
Notification, Asynchronous Execution of Tasks in Web Applications, Polling, Caching and Scaling of Storage Technologies

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je provést analýzu, návrh a implementaci systému pro práci a správu notifikací v rámci projektu Gloffer, které budou vznikat na základě uživatelských akcií vykonaných ve webové aplikaci. Práce bude popisovat kroky ke zvýšení výkonu a optimalizace ukládání, získávání a zpracování dát.

1. Student nastuduje problematiku systému notifikací ve webových aplikacích s možností rozšíření pro mobilní aplikace a porovná existující služby a technologie (Google Firebase, OneSignal, Redis Messaging,...).
2. Student implementuje systém notifikací pro webové aplikace. Analyzuje způsob ukládání a možnosti implementace s rozšířením pro podporu mobilních aplikací a porovnáním vlastního řešení s externími službami.
3. Součástí práce bude návrh a vytvoření rozhraní asynchronních událostí a následným hromadným zpracováním ve webových aplikacích využitím systému RabbitMQ. Cílem je implementovat způsob vyvolávání notifikace ve webové aplikaci.
4. Součástí práce bude vytvoření API rozhraní pro získávání vyhledávaných dat nad systémy ElasticSearch a Redis. Rozhraní poskytne možnost dynamického získávání dat mezi různými typy úložišť a jejich validací. Cílem je znovu použitelnost výsledků a mezivýsledků vyhledávání.
5. Student porovná možnosti ukládání a dotazování dat mezi různými typy NoSQL databází zaměřených na sloupcové a dokumentové uložení nebo typ key-value (klíč-hodnota), jako například Cassandra, MongoDB – ElasticSearch, Redis..., a analyzuje možnosti rozšiřování a škálování.
6. V závěru student provede srovnání dosažených výsledků s existujícím řešeními a zhodnotí možnosti škálování aplikace do budoucna se zaměřením na efektivitu jednotlivých řešení.

Seznam doporučené odborné literatury:

[1] GORMLEY, Clinton a Zachary TONG. Elasticsearch: the definitive guide. ISBN 1449358543.
[2] SHKLAR, Leon. a Rich. ROSEN. Web application architecture: principles, protocols and practices. 2nd ed. Hoboken, NJ: Wiley, c2009. ISBN 047051860x.
[3] SHIVAKUMAR, Shailesh Kumar. Architecting high performing, scalable and available enterprise web applications. ISBN 9780128022580.
[4] WEERAWARANA, Sanjiva. Web services platform architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more. Upper Saddle River, NJ: Prentice Hall PTR, c2005. ISBN 0131488740.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radoslav Fasuga, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 30. Apríla 2019 .....

Rád by som sa poďakoval za podporu a pomoc najmä vedúcemu diplomovej práce Ing. Radoslavovi Fasugovi Ph.D., za dohľad a vedenie vypracovania počas konzultácií.

**Abstrakt**

Diplomová práca sa zaoberá analýzou, návrhom a implementáciou systému pre prenos správ. Uvedený systém predstavuje architektúru asynchrónneho spracovania dát a zníženia závislostí medzi jednotlivými službami. Okrem danej problematiky, ktorá tvorí hlavnú časť diplomovej práce boli spracované poznatky pre ukladanie dát z hľadiska porovnania vybranných nerelačných databáz, možnosti škálovania konkrétnych technológí a ukladania výsledkov dotazov medzi nezávislými úložiskami. Práca obsahuje popis existujúcich riešení spolu s porovnaním výsledkov implementácie, ktorá je založená na konkrétnom riešenom príklade.

**Kľúčové slová**: webové aplikácie, notifikácie, správy, asynchrónne spracovanie, integrácia systémov, nerelačné databáze, cache, škálovanie technológií

**Abstract**

Diploma thesis focuses on analysis, design and implementation of system for transferring messages. The solution represents an architecture of asynchronous data processing and decoupling dependencies between individual services. Beside the main scope of the thesis, there is also processed knowledge about data storing techniques from comparison of selected non-relational databases, possibilities of scaling these technologies and storing results of queries among independent data storages point of view. There is a description of existing solutions with a comparison of implementation results which is based on given example.

**Key Words**: web applications, notifications, messages, asynchronous processing, systems integration, non-relational databases, cache, technologies scaling

# Obsah

# Zoznam použitých skratiek a symbolov

| | | |
|---|---|---|
| JIT | – | Just-in-time |
| REST | – | Representational State Transfer |
| JSON | – | JavaScript Object Notation |
| API | – | Application Programming Interface |
| HTTP | – | Hypertext Transfer Protocol |
| HTTPS | – | Hypertext Transfer Protocol Secure |
| STOMP | – | Simple Text Oriented Message Protocol |
| MQTT | – | Message Queuing Telemetry Transport |
| AMQP | – | Advanced Message Queuing Protocol |
| NRT | – | Near Real Time |
| CRUD | – | Create, Read, Update, a Delete operácie |
| XML | – | Extensible Markup Language |
| CSV | – | Comma-separated Values |
| STC | – | Suffix Tree Clustering |
| CQL | – | Common Query Language |
| RDBMS | – | Relational Database Management System |
| JDBC | – | Java Database Connectivity |
| HDFS | – | Hadoop Distributed File System |
| PAM | – | Pluggable Authentication Module |
| BSON | – | Binárne enkódovaný JSON |
| GUI | – | Graphical User Interface |
| ACID | – | Atomicity, Consistency, Isolation, Durability |
| LDAP | – | Lightweight Directory Access Protocol |
| TCP | – | Transmission Control Protocol |
| LRU | – | Least Recently Used |
| JVM | – | Java virtual machine |
| SDK | – | Software development kit |
| XMPP | – | Extensible Messaging and Presence Protocol |
| FCM | – | Firebase Cloud Messaging |
| PUB/SUB | – | Publish–subscribe vzor |
| SQL | – | Structured Query Language |
| NoSQL | – | Označenie pre nerelačné databáze |
| RAM | – | Random Access Memory |
| SSD | – | Solid-state drive |
| IRC | – | Internet Relay Chat |
| RSS | – | XML formátovaný text obsahujúci webové príspevky |

| | | |
|---|---|---|
| CDF | – | Channel Definition Format |
| W3C | – | World Wide Web Consortium |
| MIME | – | Multipurpose Internet Mail Extensions |
| WHATWG | – | The Web Hypertext Application Technology Working Group |
| APN | – | Apple Push Notification |
| FIFO | – | First in - first out |
| URL | – | Uniform Resource Locator |
| TTL | – | Time-to-live |
| JWT | – | JSON Web Token |
| OAuth | – | Otvorený štandard pre riadenie prístupu |
| CQRS | – | Command Query Responsibility Segregation |
| TLS | – | Transport Layer Security |
| LFU | – | Least Frequently Used |
| DB | – | Databáza |
| CGI | – | Common Gateway Interface |
| IETF | – | Internet Engineering Task Force |
| P2P | – | Peer-to-peer |
| FBP | – | Flow-based programming |

# Zoznam obrázkov

# Zoznam tabuliek

# Zoznam výpisov zdrojového kódu

# 1 Úvod

Práca sa zaoberá analýzou, návrhom a implementáciou návrhového riešenia systému, ktorý ponúkne nadhľad do problematiky asynchrónneho spracovávania úloh a práce systémov využívajúcich na prenos dát a informácií o udalostiach takzvané správy pričom uvedená téma predstavuje hlavné zameranie diplomovej práce. Samotná práca systému bude zobrazená na príklade, ktorý predstavuje automobilový inzerentný portál využívajúci webové notifikácie pre svojich používateľov. Okrem toho práca obsahuje poznatky o nerelačných databázach, ukladaní dát, škálovaní technológií, ktoré tvoria dodatočný informačný prínos z hľadiska témy. V úvodnej kapitole sú zobrazené prehľadne vlastnosti vybraných nerelačných databáz s následným porovnaním z hľadiska možností škálovania a dotazovania so zámerom na získanie relevantných výsledkov využitím textovéj formy vyhľadávania. Obsahom sú taktiež popísané existujúce riešenia zasielania push notifikácií pre webové a mobilné zariadenia. V ďalšej kapitole je popísaný princíp architektúry prenosu správ medzi systémami, vlastnosti jednotlivých častí a zobrazenie procesu od odoslania až po doručenie konkrétnemu prijímateľovi spolu s možnosťami monitorovania chýb. Popísané poznatky boli následne využité v ďalších kapitolách, ktoré sa zaoberajú implementáciou riešení zo zadania diplomovej práce. V prvom prípade to je zasielanie webových notifikácií využitím externej služby, následne návrh vlastného riešenia, ktoré spĺňa opísané funkčné požiadavky a jeho implementácia. Poslednú časť implementácie tvorí rozhranie na uchovávanie dát medzi nezávislými úložiskami. Jednotlivé kapitoly implementácie obsahujú sumarizované údaje o dosiahnutých výsledkoch a porovnaní vlastností.

# 2 Technológie pre ukladanie a prácu s dátami

Úvodný popis jednotlivých technológií obsahuje kľúčové vlastnosti pre jednotlivé riešenia spolu s určením, pri ktorom vieme z konkrétnej technológie využiť jej hlavné výhody pre koncové riešenie. Samotné technológie, postupy či protokoly a ich prípadné použitie pri vývoji konkrétnych riešení budú analyzované v ďalších častiach práce.

## 2.1 Vlastnosti vybraných technológií

Vybrané technológie predstavujú skupiny, medzi ktoré zaraďujem dokumentovo orientované databáze zamerané na full-textové vyhľadávanie, ukladanie veľkého množstva dát (BigTable) spolu s Key-Value databázami a v neposlednom rade cloud služby pre real-time ukladanie dát a notifikácie vo webových aplikáciách.

### 2.1.1 Elasticsearch

Technológia využívaná vďaka možnostiam ako pracovať s dátami pri použití full-textového vyhľadávania. NRT platforma pracuje v takmer reálnom čase (latencia do 1s odkedy sa zaindexuje dokument a je ho možné vyhľadať). Rozdelenie inštancií je v rámci klasterov, ktoré obsahujú jednotlivé serveri (nodes). Kolekcia dokumentov, ktoré predstavujú konkrétne dáta určitého typu je uložená v indexe. Pre rozdelenie indexu na viac uzlov sú využívané shardy a repliky.

Dotazovanie prebieha cez REST API rozhranie (typické CRUD operácie) a ďalej ponúka interné API pre indexovanie, update, delete operácie či ďalšie nad samotným indexom. Výsledky z vyhľadávania je možné ďalej agregovať a podľa presne zadaných metrík z dotazu sa dá nad samotnými dátami aj filtrovať.

Rozhranie X-Pack ponúka prvky pre zabezpečenie celého klasteru a je schopné odosielať stavy z monitoringu a taktiež aj dodatočný machine-learning nad dátami so zobrazením výstupu pomocou rôznych grafov. Testovanie klasteru a celej infraštruktúry prebieha pomocou Java Testing Frameworku. Predspracovanie dokumentov prebieha v rámci ingest-node rozhrania, ktoré pomocou pipeline reťazí postupne jednotlivé procesy presne v poradí ako budú spracované dokumenty.

Pracuje s dostupnými typmi, pre mapovanie dokumentov, medzi ktoré patria napríklad číselné hodnoty, reťazce či objekty a geo typy. Jednotlivé definované stĺpce sú zdieľané v rámci mappingu, ktorý je nad indexom. Pri neexistujúcom type pri použití dynamického mappingu dochádza k odvodeniu typu zo spracovávaných dát. Vizuálne real-time zobrazenie dát a ich analýzu ponúka Kibana, webové rozhranie s rozličnými grafmi.

Logstash realtime pipeline slúži pre spracovanie dát, kde dynamický unifikuje dáta pre analýzu, monitoring. Dáta môžu predstavovať typické logy (webové logy Apache, aplikačné log4j Java) či transformáciu http dotazov na eventy. [1]

### 2.1.2 Apache Solr

Pre ukladanie dát ponúka (schemaless) typ uloženia s aplikovaním schema nodes, pravidlami, ktoré definujú následne typ uložených dát. Automatické mapovanie stĺpcov prebieha podľa názvu poľa príslušných dát. Spracovanie dát je takmer v reálnom čase. Pre integráciu so serverom sa používa REST rozhranie a pre prácu s klientom sú dostupný natívni klienti pre jednotlivé prostredia spolu s open source projektami vyvíjanými komunitou. Rozličné typy pre ukladané dát ponúkajú rozšírené možnosti ako sa nad nimi dotazovať a filtrovať.

V rámci vyhľadávania nad dokumentami sa využíva hĺbkový kurzor a pre agregované skupiny (facety) sa dajú aplikovať vyhľadávacie podmienky v rámci dotazu. Rozšírené vyhľadávanie pre formy ako nápoveda nad dátami, kontrola pravopisu a zvýraznenie výsledkov vo vyhľadávaní. Nad dátami je možne následne prevádzať rôzne štatistiky, zoraďovať dokumenty vo výsledku, nastavovať prioritu konkrétnych dát a výpočtov pre agregácie. Priebeh operácií pozostáva z indexu, ktorý je vytvorený pri vložený samotných dát. Ten predstavuje formát ako sú ukladané dáta. Následne sa využívajú dotazy pre získanie výsledkov, ktoré sú transformované do podoby mapovaním na uložené dokumenty a výstup je následne pomocou algoritmov vyhľadávanie ohodnotený a podľa toho vložený na výstup. Medzi podobné vlastnosti NoSQL patria facety, ktoré sú využívanie ako forma agregácie pre získanie výsledku. Na základe vyhľadávacích podmienok – kritérií sú výsledky rozdelené do podskupín. Automatická nápoveda výsledkov prebieha na základe vstupného reťazcu, ktorý je ohodnotený použitými algoritmami a na výstupe sú výsledky, ktoré by mu mohli zodpovedať. Ďalšími možnosťami vo vyhľadávaní sú kontrola gramatiky, zvýraznenie vyhľadávaných reťazcov vo výsledkoch a geo dotazy.

Mimo už uvedené možnosti vyhľadávanie tu sú aj rozšírenie pre priestorové dáta (filtrovanie dát na základe polohy), výpočty zložitých vzorov, geografických dát a integrácia s Java Topological Suite Library. Solr podporuje formáty JSON, XML, CSV a ďalšie pre prácu s dátami. V rámci cache je možné ukladať dotazy, filtre a samotné dokumenty.

Škálovanie pomocou technológie Apache Zookeper, vytváranie transakčných logov, replík a failover scenárov, rozdelenie dát do shardov bez potreby reindexácie, vstavané používateľské rozhranie. Samotná konfigurácia je centrálna, zatiaľ čo index je distribuovaný. Plán v prípade zlyhania je plne automatický a pri zasielaní dotazov je možný taktiež load-balancing, výber uzlu na ktorý sa dotaz zašle.

Administračné rozhranie ponúka prehľad o celom systéme a pri aplikovaný konkrétnych dotazov aj prehľad o dátach. Klaster, ktorý sa stará o výsledky vyhľadávania je založený na technológií Carrot2. Tá automaticky vytvára z menších kolekcií dát klástre, taktiež chápané ako výsledky, ktoré sú vrátené pri dotazovaní a získavaní dokumentov a predstavujú určitú úroveň abstrakcie, keďže sú to kategórie dát. Pre klastering sú štandardne 2 algoritmy, Lingo (založené na dekompozícií hodnôt) a STC, ktorý predstavuje formát trie, suffixových stromov. [2]

### 2.1.3 Apache Cassandra

Technológia vyvíjaná spoločnosťou Apache Software Foundation spoločné s ostatnými technológiami určenými najmä pre ukladanie a prácu s dátami. Hlavnou výhodou je správa veľkého objemu dát s možnosťou škálovania architektúry s využitím prvkov NoSQL. Dáta sú automatický replikované medzi jednotlivými uzly. Využitie nachádzame najmä vďaka vlastnostiam, ktoré sa týkajú návrhu architektúry, konkrétne decentralizovaná architektúra. Výkon je vysoko škálovateľný vďaka lineárnemu princípu čo predstavuje zvýšenie výkonu pridaním nových uzlov. Jednotlivé zmeny sú dynamicky spracované v rámci systému. Systém je vyvíjaný ako open-source.

Výkon zápisu a čítania sa lineárne zvyšuje pridávaním nových uzlov a záruka uchovania dát je aj pri výpadkoch jednotlivých serverov. Cassandra pracuje v vlastným dotazovacím jazykom CQL, ktorý má podobnú syntax ako SQL. Pre prácu s dátami sa používa formát JSON. Dáta sú ukladané v rámci kľúčových priestorov (keyspace) v tabuľkách, je nad nimi možné vykonávať CRUD operácie, vytvárať indexy či materializovaný pohľady. Rýchle zápisy je možné dosiahnuť aj pri použití menej výkonného hardwaru a s možnosťou uchovania stoviek TB. Okrem natívnych dátových typov ako integer, string a ďalšie je možné pracovať s dátami typu (list, set a maps), ktoré sú určené pre uchovanie menšieho počtu denormalizovaných dát. Ponúka možnosť definovať vlastné dátové typy (podobnosť triedy v programovacích jazykoch) a existuje podpora pre širokú škálu programovacích jazykov v podobe klientskych ovládačov.

Replikácie môže prebiehať v móde master-master a jednotlivé uzly pracujú v vlastným úložiskom. Cassandra implementuje štýl Dynamo replikácie, ktorá neobsahuje žiaden bod, ktorý by v rámci klasteru spôsobil zlyhanie. Výhody oproti relačnému databázovému systému sú najmä pri použití neštruktúrovaných dát. Samotná schéma je oveľa flexibilnejšia. Tabuľka predstavuje vnorený zoznam kľúčov a hodnôt s rozšírením na stĺpce. Jednotlivé riadky a stĺpce predstavujú najmenšiu jednotku pre ukladanie zatiaľ čo pri RDBMS je stĺpec chápaný ako atribút. Vzťahy nie sú viazané pomocou cudzích kľúčov ale cez samotné kolekcie, v ktorých sú ukladané neštruktúrované dáta. [3]

### 2.1.4 Apache HBase

Podobne ako Cassandra patrí pod Apache Software Foundation, open-source nerelačná databáza. Primárne využitie je pre prácu s Big Data a čítaním/zápisom v takmer reálnom čase. Hlavnou úlohou je uchovávať veľké množstvo dát vo veľmi veľkých tabuľkách. Škálovateľnosť je lineárna a modulárna.

Pre jednotlivé tabuľky je možné vytvárať automatické shardy s možnosťou ďalšej konfigurácie. Jednotlivé uzly (serveri) môžu byť usporiadané v rámci regiónov čo prináša failover riešenie v prípade výpadku niektorej zo skupín. Populárna práca s dátami je využitím Hadoop MapReduce technológie. Dotazy, ktoré sú prevádzané v takmer reálnom čase môžu byť uchovávané v rámci blokovej cache.

Podporuje REST pomocou XML a možnosťou binárneho kódovania dát. Metriky je možné ďalej exportovať cez Hadoop Metrics prípadne službu Ganglia. V ponuke je natívny JAVA klient.

Jednotlivé riadky, ktoré sú spoločne v jednej tabuľke môžu byť ukladané v rámci column-family a samotné tabuľky sú umiestňované v menných priestorov (namespaces) a patria do určitej schémy. Oproti uvedenej Cassandre je lepšie podpora škálovania (pri 1000+ jednotkách). V rámci ukladania je využívaná kompresia jednotlivých dát a spracovanie prebieha najmä formou in-memory. Nad stĺpcami pracujú Bloom filtre. Tie predstavujú pravdepodobnostnú dátovú štruktúru čoho výsledkom je otestovanie či je konkrétny záznam uložený v stĺpci. Výsledkom spojenia jednotlivých dát a stĺpcov je BigTable.

Samotná databáza nie je založená ako náhrada za klasický SQL databázový systém. Jednotlivé analytické a štatistické funkcie môžu byť využívané použitím napríklad JDBC driver. Konzistencia pre čítanie a zápis nie je však prioritou HBase, ktorá sa zameriava na vysokorýchlostné agregačné dotazy. Pre prípady, kedy náš use-case obsahuje tisícky alebo niekoľko miliónov záznamov môže byť lepšia voľba použiť klasický RDBSM systém. Dátový systém je postavený and HDFS, ktorý dokáže efektívne pracovať s veľkými súbormi pokiaľ sú práve distribuované. Oproti tomu vytvára indexované StoreFiles, ktoré sú ako vrstva and HDFS vyhľadávaním. Zabezpečenie samotných dát je realizované pomocou Zookepera a HDFS aby samotný používateľ nedokázal pristupovať k dátam a metadátam. Použitím Zookepera vieme vytvárať ACL zoznamy pre obmedzenie prístupu. Samotné zabezpečenie HDFS sa spolieha na vrstvu súborového systému. Ďalej je možné pracovať s role-based autentifikáciou, visibility labels pre označenie zdrojov, ktoré majú byť zabezpečené či transparentným šifrovaním dát.

Medzi ďalšie podmienky, ktoré je vhodné splniť pre používanie je minimálne počet uzlov, ktorý je odporúčaný v minimálnom počte 5. Aplikácia, ktorá využíva vlastnosti RDBMS ako transakcie, triggre či komplexné join operácie nie je vhodná práve vhodná pre využívanie HBase ako primárneho dátového úložiska. Výhoda vzniká pri variabilite samotnej schémy s menej odlišnými typmi dát v rámci jednotlivých riadkov a pri použití kľúčov, ktorými sa odkazujeme na uložené dáta. [4]

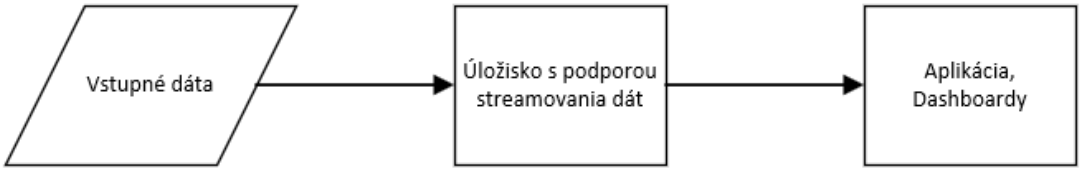
### 2.1.5 MemSQL

Oproti uvedeným technológiám je MemSQL jedna z najmladších, kedy prvé vydanie bolo v roku 2013. Hlavným prvkom je SQL databáza, ktorá ponúka vysoký výkon hlavne pri analýze dát s veľkým počtom konkurentných používateľov. Môže fungovať v rámci cloudu a pracovať s PB dát.

Dáta sa spracované na výstupe cez jednotlivé pipelines (streamovanie dát). Pre historické dáta je možnosť kompresie pri uložení. Plnenie dát môže byť prevádzane zo zdrojov ako Apache Kafka, Amazon S3 ale taktiež aj z iných databáz ako MySQL, SQL Server či Oracle Database. (Schéma 1)

Pre prácu sú dostupné nástroje pre business intelligence a machine learning. Dáta sú ukladané vo formáte JSON s podporou geografických typov (geografické koordináty, vzdialenosti,...).

Optimalizácie dotazov sú schopné prebiehať na úrovni strojového kódu a v rámci jednej transakcie. Spájanie dát (JOIN) medzi tabuľkami je distribuované pre jednoduchší prístup. Ukladanie pri paralelnom spracovaní môže prebiehať medzi viacerými uzlami (distribuované úložisko). Historické dáta sú ukladané na disk zatiaľ čo jednotlivé streamy sú uchovávané v pamäti. Pre monitorovanie existujú dostupné nástroje s prehľadom nad klastrom a ďalej je možné konfigurovať bezpečnosť na základe používateľských práv, skupín, šifrovať dáta, PAM autentifikácia, logovanie auditov a akcií nad klastrom. [5]




Obr. 1: Zobrazenie stream časti MemSQL

### 2.1.6 MongoDB

Kombinácia kritických prvkov relačných databáz spolu s nerelačnými je základom pre uvedenú NoSQL databázu. MongoDB pracuje s vlastným dotazovacím jazykom, ponúka vysokú flexibilitu a konzistenciu dát. Dostupné možnosti pre škálovanie a zvyšovanie tak výkonu pri práci s dátami aj vďaka viacmodelovej architektúre, kde dátový model reprezentuje dáta cez jednoduchý odkaz pomocou kľúča k hodnote.

Formát dát pre uložené dokumenty je BSON (binárne enkódovaný JSON) spoločne s denormalizovaným tvarom dát, ktorý je častým prvkom pri NoSQL databázach. Každý dokument má priradený unikátny kľúč a dokument predstavuje samostatný objekt. Dostupný nástroj MongoDB Compass ponúka GUI rozhranie a možnosť prevedenia dotazovacích operácií nad úložiskom. Okrem tradičných dotazov pre zisk dát (rozsahový dotaz, vyhľadávanie) podporuje spájanie (JOIN) viacerých kolekcií, agregačné dotazy, MapReduce dotazy pre spracovanie dát v databáze. Vstavaný súborový systém je nazvaný GridFs, podobný HDFS pri Hadoop systéme. Štandardný limit pre ukladanie je 16 MB pre jeden dokument. Oproti ostatným technológiám nemá takú silnú väzbu na ďalšie nástroje. Okrem MapReduce je pre agregácie možné použiť agregačný pipeline, ktorý dosahuje lepšie výsledky z hľadiska výkonu.

Lineárne škálovanie dát je zabezpečené automatickým shardingom, distribúcia dát medzi viacerými servermi (fyzickými partíciami), možnosť definovať rozsah kľúčov, podľa hashu hodnoty a podľa zóny umiestnenia dát. Pri škálovaní stále pracujeme s jedným primárnym uzlom a ďalšími, ktoré sú sekundárne. Replikácia predstavuje master-slave topológiu. Dostupnosť je tak zaručená prvkami, ktoré pracujú s uzlami v prípade výpadku alebo inej zmeny. Podpora ACID

transakcií bola pridaná v novšej verzií roku 2018 a izoluje operácie použitím snapshotov, ktoré sa aplikujú ako rollback v prípade neúspechu transakcie.

Vizualizácia dát je možná cez MongoDB Connector for Business Intelligence. Pri použití vhodnej kompresie je možné dosiahnuť ušetrenie o takmer 70%. Bezpečnosť môže byť zabezpečená použitím overovaním cez LDAP, Windows Active Directory, autorizácie používateľských rolí a ďalej využiť audit prístupov a šifrovanie dát. Pre manažment správy služieb ponúka MongoDB prostriedky použitými webovými nástrojmi, ktoré sledujú aktivitu samotné hardwaru na ktorom je spustený systém. Ďalej to sú upozornenia na nájdene chyby a konzola pre optimalizáciu výkonu.

Okrem bežných dotazov pre získanie výsledkov z dokumentov tu pracuje aj facet vyhľadávanie, ktoré využíva agregačné dotazy a pridáva možnosť vyhľadávať nad dátami podľa určitých vlastností. Filtrovanie a zoradenie dát pomocou MapReduce (produkuje veľký objem dát pre spracovanie), reduce pre výsledok sumarizovanej operácie. Indexovanie prebieha nad jednotlivými dokumentami, primárna aj sekundárne úložiská.

Zaujímavým prvkom je aj serverless platforma, ktoré je určená pre aplikačný vývoj. Ten predstavuje architektúru kedy sa klient pomocou vstavaných funkcií využíva službu. Pod označením stitch ponuka Mongo služby pre dotazovanie, spúšťanie javascriptových funkcií v rámci služby, triggre pre zaslanie notifikácií o zmenách v dabaze a synchronizáciu dokumentov, ktoré sú ukladané lokálne v zariadení a časťou backendu v reálnom čase. [6]

### 2.1.7 Redis

Open source ukladanie štruktúrovaných dát v pamäti, najčastejšie použité práve ako cache pre dáta ale taktiež možnosť využiť ako "databázu"či PUB/SUB systém pre správy. Nechýba viacero dátových typov (string, hash, list, set, zoradený set, bitmapy, geografické dáta).

Replikácia prebieha v móde master-slave, kde slave inštancie obsahujú kópiu master uzlu. Pre zložitejšie dotazy nad úložiskom je možnosť použiť vstavaný LUA skript interpeter. Dáta môžu byť uchované na disku v podobe snapshotov taktiež aj v podobe zálohy. Operácie ako pripojenie k reťazcom, zvýšenie hodnoty podľa hash kľúču, vkladanie hodnôt do zoznamu, výpočet prieniku, zjednotenia a rozdielu množín, výber prvku s najvyššou hodnotou či iné zoradenie prebiehajú atomicky. Najjednoduchším spôsobom pre partitioning je využitie rozsahov, kedy mapujeme rozsah dat pre každú inštanciu a na základe toho sú smerované dotazy. Oproti tomu existuje aj hash partitioning, kde pracujeme s kľúčom, ktorý je následne napríklad pomocou operácie modulo výsledkom pre nájdenie konkrétnej inštancie, kde sú uchované dáta.

Systém pracuje s rozličnými príkazmi pomocou ktorých sa dá pracovať. V prípade MULTI príkazu dokáže Redis prevádzať transakcie nad viacerými príkazmi v rámci jedného behu. Jednotlivé príkazy v rámci transakcie sú serializované a spúšťané samostatne. Pri spracovaní Redis pracuje plne atomicky. Samotný systém však nepridáva možnosť rollback operácie nakoľko zlyhanie dotazu je skôr spôsobné chybnou syntaxou alebo chybou na strane klienta – programátora. Zámok nad jednotlivými kľúčmi prebieha v rámci monitorovania, ktoré je počas transakcie. V

prípade, že sú monitorované dáta zmenené, celá transakcia je ukončená. Pre dáta priradené konkrétnemu kľúču je možné nastaviť time-to-live kedy budú zmazané. Nechýba podpora rozličných klientov pre programovacie jazyky. Pre hromadné vkladanie je možné vygenerovať takzvaný Redis protokol, ktorý predstavuje postupnosť jednotlivých príkazov avšak zapísaných v konkrétnom súbore. Nahratie súboru je následne možné pomocou Redis CLI, pipe, ktorý odošle ihneď dáta na server.

Komunikácia prebieha pomocou protokolu TCP, vykonanie viacerých príkazov naraz je pomocou pipeline. Serveri môžu byť rozdelené na rozličné partície s určitou konfiguráciou veľkosti úložiska a priradenými systémovými prostriedkami. V prípade výpadu master jednotky sa dá špeficikovať failover plán. Pre ušetrenie miesta v rámci ukladania sú použité práve vstavané dátové typy uvedené vyššie, ktoré sú navyše optimalizované – kódované. Vďaka tomu je možné vyladiť využitie procesoru či pamäte. Nastavenia sa týkajú veľkosti ukladaných hodnôt a počtu kľúčov pre typy ako set, list a hash. Práve hash funguje v rámci Redisu ako jeden z najefektívnejších typov pre ukladanie. V prípade, že reprezentujú objekty, ktoré pozostávajú z viacerých atribútov vieme použitím optimálnej veľkosti v závislosti na hardwari ušetriť priestor potrebný pre uloženie a dosahovať zložitosť vyhľadávania O(1).

Pri ukladaní Redis automaticky spravuje cache formou LRU cache. Staré dáta, ktoré boli vyhodnotené pomocou pravidiel sú v prípade nedostatku zdrojov zmazané z pamäte. Zmazanie môže prebiehať podľa viacerých princípov LRU, kedy sa žiadne nemusia zmazať, sú zmazané náhodne, LRU – najmenej používané alebo tie, ktoré čoskoro exspirujú. Samotný algoritmus sa dá konfigurovať napríklad nastavením počtu prvkov, ktoré sa testujú a tak získať lepšiu presnosť. [7]

## 2.2 Škálovanie a zvyšovanie dostupnosti

V rámci škálovania a zvyšovania dostupnosti, ktoré môžu vznikať v prípade vyššieho počtu operácií za sekundu prípadne potreby ukladať väčšie množstvo dát ponúkajú dané vyhľadávacie technológie viacero úrovni nad ktorými sa dá zvyšovať výkon. Pre všeobecný popis som vybral technológie Elasticsearch a MongoDB s dodatočným popisom jednotlivých vlastností, ktoré sa týkajú predmetu škálovania.

Výhodou ElasticSearch aj vďaka princípu NoSQL je vlastnosť spracovať aj veľké množstvo dát, ktoré sú potrebné pre indexovanie. V prípade nedostatkov je možné pridávať ďalšie uzly a vytvárať tak shardy, ktoré obsahujú časti dát v rámci klasteru či replicy ako záložné uzly v prípade výpadku. Jednoduchý prechod na danú technológiu zaručuje aj engine Lucene, nad ktorým je postavený Elasticsearch. Ten zahŕňa najmä prvky ako formát dát, spôsob indexovania a aj samotný výkon.

Rozličný typ dát, ktorý sa ukladá do databáze taktiež nie je problémom. Častou požiadavkou je totiž aj samotná prac na miliónmi / miliardou dokumentov a samotné zaslané dotazy sa môžu líšiť. Podmienkou je taktiež rýchlosť odpovede, ktorá môže byť prijateľná v jednotkách milisekúnd či niekoľko desiatkach.

Menším problémom, ktorý môže vznikať väčším počtom dát sú zdroje. Pamäťový priestor je v tomto prípade spravovaný JVM (Java Virtual Machine) a v prípade cachovanie výsledkov – dokumentov môže dochádzať k rýchlejšiemu vyčerpaniu zdrojov aj v prípade použitia niekoľko GB operačnej pamäte. Nastavenia pre ukladanie - cachovanie sa však dajú nastaviť ako aj samotný dostupný priestor.

Pre zaručenie failover plánu je vhodné spustiť samotný master node na dedikovanom uzly. Samotný uzol sa tak stará o alokáciu jednotlivým shardom, mapuje ukladanie replík a prevádza preusporiadanie s vysokým výkonom. Dotaz môže byť smerovaný na ľubovoľný uzol vďaka internému smerovaniu. Samotný index môže byť rozdelený medzi ľubovoľný počet shardov a k tomu ďalšie repliky.

Podobne to je ako v ostatných technológiách ako napríklad MongoDB kde je taktiež možnosť začať škálovanie z jedného servera až na klaster o veľkosti 1000 uzlov. Pre lepšiu špecifikáciu klasteru môžeme uvažovať ako o distribuovanej databáze, ktorá sa rozkladá na x fyzických servoch a rôznych lokalitách, najčastejšie v rôznych dátových centrách. Výkon je možné škálovať aj za hranice 100 000 read/write operácií za sekundu a uchovávať miliardy dokumentov. [8]

### 2.2.1 Vertikálne škálovanie

V tomto prípade sa snažíme zvyšovať výkon jednotlivého serveru. Môže to byť dosiahnuté pridávaním systémových zdrojov ako pamäť RAM či výkonnejší procesor. V prípade cloudových riešení však môžeme naraziť na strop, kedy už nebude možné pridávať dodatočné hardwarové prostriedky.

### 2.2.2 Horizontalne škálovanie

Samotné dáta a spracovanie sú rozdelené medzi viacero fyzických serverov. V prípade nedostatku kapacity sa pridajú nové zariadenia. Aj v prípade, že výkon jednotlivých serverov nemusí byť tak vysoký ako v prípade vertikálneho škálovania, samotný workload je najčastejšie vďaka technológiám rovnomerne rozdelený a je možné dosiahnuť lepšiu efektivitu. Nevýhodou je zložitejšia architektúra a spravovanie klasteru.

### 2.2.3 MongoDB

V prípade shardingu môžeme celkové dáta rozdeliť ma menšie celky, ktoré sú umiestnené na serveroch. Zo shardov následne môžeme vytvoriť repliky. Repliky narozdiel od shardov obsahujú rovnaké dáta. Zabezpečujú najmä redundanciu a dátovú dostupnosť. V niektorých prípadoch taktiež aj zvýšený výkon v prípade čítania dát. Medzi aplikáciu a kláster zostavený zo shardov je možnosť umiestniť Mongos query router, ktorý dokáže sledovať metadáta o dátach, ktoré sú uchovávané na jednotlivých shardoch. Zdroje na udržiavanie smerovača sú minimálne a uchováva si perzistentný stav. Samotná konfigurácia je uchovaná v config serveroch, ktoré sú v novších verzia databáze nasadzované taktiež ako repliky.

Smerovanie požiadavkou na klaster začína v samotnej aplikácií a mongos router ďalej zabezpečuje zistenie shardov, ktoré obsahujú dáta a ktorým predá ďalej dotaz. Dokáže zostaviť kurzor, kedy výsledky jednotlivých shardov sú spracované na primárnom uzle a vrátené ako jeden. Primárny uzol môže taktiež zabezpečovať operáciu pre zoradenie, prípadne nastavenie veľkosti výsledku a agregáciu čo však nie je podmienkou vykonávania práve na primárnom uzly. [9]

### 2.2.4 Apache Solr

Pre zvýšenie dostupnosti a toleranciu chýb je dostupný SolrCloud, ktorý rieši distribúciu samotného indexovania a vyhľadávanie medzi viacerými uzlami. Konfigurácia je centrálna s automatickým rozkladom záťaže. Koordinácia môže byť spravovaná použitím ZooKeepera. Využíva podobné techniky pre tvorbu shardov, replík a smerovania dotazov na základe ID prefixu. V prípade nedostupnosti môže samotný dotaz automaticky zlyhať prípadne obdržať výsledok v stave kedy pre každý shard existuje minimálne jedna funkčná replika. Výsledok však nemusí byť presný. [10]

### 2.2.5 Apache Cassandra

Oproti spomenutým technológiám ponúka mierne rozšírený pohľad na to ako replikovať samotné dáta a využíva pritom Cassandra Dynamo. Replikácia môže byť zvolená medzi 2 dostupné stratégie. Pri jednoduchej definujeme počet uzlov, ktoré obdržia kópiu každého riadku zatiaľ čo pri sieťovej stratégií definujeme replikáciu samostatne pre každé data centrum v klastri. Pre konzistenciu je možné definovať úrovne, ktoré zodpovedajú úspešnému stavu výsledku ako v prípade kedy musí odpovedať iba jedna replika alebo všetky.

Dynamo koncept predstavuje 128 bitový kľúč pre partíciu a všetky možné hodnoty, ktoré s ním súvisia, formujú kruh (ring) a každý uzol v rámci klastru zodpovedá pre jeden alebo viac rozsahov v rámci okruhu. Vďaka hashovaniu tak vieme zistiť samostatné rozsahy dát. [11]

### 2.2.6 MemSQL

Jednoduchšia architektúra s ohľadom na priamočiarosť a výkon, ktorý je zabezpečený vďaka distribuovanému systému. Samostný klaster je rozdelený na dve úrovne. Prvou z nich sú agregátori a ďalšou listové uzly, ktoré spracúvajú čo najviac dát, spúšťajú SQL dotaz na základe výsledkov z agregátoru. Listový uzol môže pozostávať z viacerých partícií. Agregátor sa stará o uchovanie metadát, smerovanie dotazov a agregovanie výsledku. Tabuľky sú replikované medzi oboma úrovňami.

Data sharding je prevádzaný automaticky vytvorením hashu na základe hodnoty primárneho kľúču. Každá partícia má určitý rozsah hash hodnoty a samotne predstavuje databázu na listovom uzly. Druhotné indexy sú spravované v rámci nej. Pre vyššiu dostupnosť je možné

konfigurovať skupiny, ktoré sú v podstate množiny listových uzlov, ktoré uchovávajú samotné dáta. Každá skupina obsahuje vlastnú kópiu každej partície v rámci systému. [12]

### 2.2.7 Apache Hbase

Pracuje ako správca uložených dát pre Hadoop a zmenou oproti iným je zacielenie na spracovanie obrovského počtu dát, riešenie pre Big Data. Škálovanie je horizontálne a nazvané ako „Region“. Samotné regióny predstavujú podmnožiny dát z tabuľky a dáta, ktoré sú nejakým spôsobom zoradené a a ináč spracované sú ukladané spoločne. Region serveri sú zodpovedné sa poskytovanie množiny regions dát a každý spravuje práve jednu časť s celkových dát. Architektúra je rozdelená na Master – Slave, kde Master koordinuje spracovanie a Slave predstavuje Region Server.

Ako v predošlých prípadoch aj tu sa uchovávajú meta dáta o tom, kde sú ukladané dáta a ako sa k ním dostať v rámci dotazu. Na konci dotazu, sa nachádza Hadoop File system, na ktorý sa pripája región server. [13]

### 2.2.8 Redis

Pre rozloženie dát využíva data partitioning, ktorý rozkladá dáta medzi jednotlivé inštancie Redisu. Rozsahy objektov, ktoré môžu byť na základe unikátnej hodnoty sa ukladané do partícií a výsledok vychádza z mapping rozsahu. Nevýhodou je potreba uchovávania dodatočných metadát. Alternatívou je hash partitioning, kedy sa nad kľúčom dát vytvorí hash, výsledkom čoho pri zvolení napríklad crc32 algoritmu je číselná hodnota a po prevedení operácie modulo, ktorá zodpovedá počtu inštancií získame hodnotu konkrétnej z nich.

Dáta môžu byť ukladané do partície na základe zvolenia klienta, asistovanej proxy na základe konfigurácia databáze prípadne smerovania dotazu, ktorý je spracovaný inštanciou, ktorá aktuálne spracováva dotaz. Nevýhodou partícií je zvýšená zložitosť riešenia, transakcie, ktoré využívajú viacero kľúčov nemôžu byť použité (napríklad v prípade prieniku dvoch množín) a perzistencia dát.

V prípade využitia databáze ako cache odpadá potreba riešiť škálovanie nakoľko hashovanie je konzistentné zatiaľ čo v prípade ukladaní dát, kde sa určené kľúče pre odpovedajúce inštancie, musíme zaručiť presný a nemenný počet uzlov. V opačnom prípade dôjde k vyváženiu v prípade zmeny počtu. Paritioning môže byť prevádzaný automaticky využitím Redis klaster.

Spomenutá architektúra Master – Slave je využitá najmä pri replikácií. Ta prebieha plne asynchronne. Master môže mať viacero slave uzlov, ktoré dokážu komunikovať medzi sebou. Replikácia z master žiadnym spôsobom neblokuje spracovanie dotazov, podobne na ostatných, kde však môžeme konfigurovať správanie sa pri spracovaní dotazu ak prebieha synchronizácia. Pri replikácií sa ukladá pre master ID ktoré ho definuje a taktiež predstavuje verziu uložených dát. [14]

### 2.2.9 Súhrn

Z uvedených údajov vyplýva, že jednotlivé technológie využívajú podobne technologické riešenia ako vytvoriť plán pre zvýšenie dostupnosti a výkonu, ktorý je potrebný pri navýšení počtu operácií zasielaných na server. Výsledkom je distribuovaný systém, ktorý je hierarchicky rozdelený podľa rolí pre jednotlivé uzly, ktoré spolu komunikujú. Dáta môžu byť distribuované medzi jednotlivé uzly, ktoré ďalej obsahujú repliky v prípade výpadku. Implementácia smerovania dotazu zabezpečí rýchly prístup ku konkrétnej časti dát a vo viacerých prípadov sa dá aj dodatočne konfigurovať. Nevýhodou je nutnosť dodatočnej konfigurácie a správy klastru.

## 2.3 Technológie a služby pre notifikácie vo webových aplikáciach

V moderných webových aplikáciách, ktoré pracujú dynamicky a neponúkajú len statický obsah sa začali využívať notifikácie často ako vyskakovacie okná (pop-up), ktoré upozorňujú používateľa najčastejšie na výsledky jeho akcií alebo majú informačný charakter. Pri nasadení uvažujeme nad spôsobom akým budú upozornenia fungovať, real-time upozornenia v porovnaní s frontou, ktorá sa v časových intervaloch vyprázdňuje. Dôležitým prvkom je aj škálovanie a rýchlosť spracovávania, ktorá sa líši v závislosti na zložitosti procesov v aplikácií tak aj na počet používateľov a ich akciami.

S príchodom nových štandardov a pokročilými funkciami vo webových prehliadačoch vznikajú externé služby, ktoré zabezpečujú odosielanie a doručovanie správ, ktoré následne notifikujú používateľa. Vývojár má tak možnosť implementovať vlastný systém upozornení alebo využiť externé riešenie.

### 2.3.1 Google Firebase

Služba ktorá fungovala ako chat aplikácia sa vďaka svojím technológiám a infraštruktúre premenila na real-time službu, ktorá doručuje dáta medzi používateľmi. Aktuálne patrí medzi služby spoločnosti Google a ponúka viacero rozšírení pre prenos a ukladanie dát.

**2.3.1.1 Firebase cloud messaging** Real-time prenos notifikácií s možnosťou implementácie pre web a mobilné technológie IOS, Android.

**2.3.1.2 Firebase Auth** Ponúka autentifikáciu používateľa voči serveru a rozhraním pre prihlasovanie cez sociálne služby ako Facebook, Twitter, Google či Github. Dáta je možné ukladať vo firebase databáze.

**2.3.1.3 Firebase storage** Umožňuje ukladať rozličný formát dát od používateľov ako video, audio a obrázky a možnosťou prepojenia s ostatnými službami.

#### 2.3.1.4 Realtime database

Je backend služba, ktorú cez HTTP Rest pristúp ponúka API prístup s dodatočnou ponukou knižníc (Android, iOS, Javascript, Java, Objective-C, Swift, Node.js a protokolom server-sent eventov, ktoré vyvoľavajú push notifikácie zo serveru na klienta.

#### 2.3.1.5 Firestore

Je v poradí jedna z najnovších služieb, ktorá ponúka flexibilnú databázu pre platformy mobil, web a serverový vývoj založenú na Google Cloud Platform. Spája služby ako realtime database s dodatočnou funkcionalitou autentifikácie. Ďalšími rozšíreniami sú nástroje pre testovanie (TestLab) a analytiku so sledovaním chybových udalostí, Analytics a Crash Reporting. [15]

### 2.3.2 OneSignal

Externá služba OneSignal ponúka rozhranie pomocou svojich SDK knižníc pre jednoduchú implementáciu push notifikácií pre jednotlivé platformy. Vývojár má v ponuke naviac prehľadné administračné dashboardy a REST rozhranie pre platformy Android, iOS a Web. Medzi prvky, ktoré sú naviac patria segmentácie zasielanie notifikácií, automatické či plánované odosielanie, API pre prijímanie správ a detailné nástroje pre reporting, ktoré dokážu napríklad zobraziť konverziu správ. Real-time analytika a štatistika využívania (online dashboardy) s možnosťou A/B testovania na používateľoch v závislosti na výsledkoch. [16]

### 2.3.3 Redis Messaging

Technológia určená pre cache/ukladanie dát poskytuje publish/subscribe rozhranie. Poskytuje príkazy na prácu s rozhraním, samotnú implementáciu rieši programátor. Rozdelenie na definované kanále, ku ktorým sa pripájajú používatelia. Komunikácia prebieha cez websocket.

Rozhranie PUB/SUB nemá prepojenie na úložný priestor (key space). Pre odbery z viacerých kanálov funguje pattern-matching na základe práve spracovávaných dát.

Rozšírenie Redis Keyspace Notifications (od verzie 2.8.0) pridáva možnosť odoberať správy pri udalostiach ako zmena konkrétneho kľúču, operácia vloženia, expirujúce dáta. Neponúka však možnosť spoľahlivých notifikácií, ak sa správa odošle a príjemca nieje pripojený, po opätovnom pripojení ju nedostane. [17]

### 2.3.4 Google Firebase Cloud Messaging

Spoľahlivý prenos správ (notifikácií) a zobrazenie používateľom. Možnosť doručovania podľa identifikátoru používateľa, cieľovej skupiny či kanálu. Klientske aplikácie môžu zasielať spätnú odpoveď o doručení.

Dostupné REST API a SDK knižnice pre mobilné platformy a desktop. Overovanie cez HTTP a XMPP protokoly, potrebné zabezpečené spojenie. Dostupné ďalšie služby ako Realtime Database pre ukladanie dát a prístup, Firebase Authentication pre overovanie a Google Analytics pre štatistiku využívania.

Ponúka asynchrónne doručovanie, automatické doručenie z FCM serveru alebo spracovanie správy u klienta. Správy môžu mať nastavenú prioritu a životnosť.

## 2.4 Dotazovanie pre získanie dát pomocou vyhľadávania

Pre získavanie dát v rámci vyhľadávacích technológií sa používa najčastejšie REST rozhranie. Volania môžu byť obalené implementovanými knižnicami pre jednotlivé programovacie jazyky. Pri vyhľadávaní sa aplikuje query search, filter a prípadne agregácie pre výsledok. Uvedené NoSQL databáze zobrazujú odlišné podanie formátu oproti jazyku SQL.

### 2.4.1 Elasticsearch

Vyhľadávanie pomocou query dotazu, v ktorom uvádzame množinu dokumentov, ktoré sa majú zhodovať vo výsledku dotazu.

```
GET /bank/_search
{"query": { "match_all": {} }}.
```

Výpis 1: Ukážka dotazu v Elasticsearch

Filtrovanie výsledkov dopĺňa štandardný query dotaz o filter s podmienkami. Filter je aplikovaný po získaní výsledku z časti „match“. Príklad dotazu je uvedený v prílohe A. Výsledky agregácie su taktiež vyhodnotené podľa výsledku a môžu byť pripojené k odpovedi.

### 2.4.2 Apache Solr

Oproti Elasticsearch ma nad vyhľadávacím enginom Lucene značne jednoduchšiu podobu. Podobná syntax je použitá aj pre nástroj ES Kibana.

```
q={}
```

Výpis 2: Vyhľadávanie v Apache Solr

```
q={ name: {Andrej TO Ondrej} }
```

Výpis 3: Filtrovanie v Apache Solr

```
&facet=true
&facet.field=manu
&facet.field=camera_type
```

Výpis 4: Agregácie Apache Solr v podobe facetov

### 2.4.3 MongoDB

```
http://adresa_serveru:port/nazovDB/nazovKolekcie/
"match" : { "name" : "andrej"}
```

Výpis 5: Vyhľadávanie v MongoDB

```
db.accounts.find( { "total": { $gt: 30 } } )
```

Výpis 6: Filtrovanie v MongoDB

Pre agregácie je obdobný zápis, definujeme stĺpec, nad ktorým sa vykoná agregácia a agregačnú funkciu. Uvedený príklad je v prílohe B.

### 2.4.4 Ostatné

V prípade ostatných uvedených databázy ako sú Cassandra, Hbase či MemSQL je využitý jazyk podobný SQL, ktorý obsahuje dodatočné prvky na podporu práce s nerelačnou schémou. Populárnym nástrojom pre Hbase je Apache Drill, ktorý pracuje ako dotazovací SQL nástroj s podporou pre ďalšie databáze. Mierne opačné riešenie predstavuje Redis, ktorý zavádza jednotlivé príkazy, ktoré fungujú ako funkcie s definovanými parametrami.

## 2.5 Porovnanie prípadov pre ukladanie dát a fulltext

V predošlej kapitole som popísal jednotlivé technológie pre ukladanie a vyhľadávanie dát. Nie všetky sa však hodia na riešenie jedného problém ak potrebujeme ukladať dáta. Musíme zvážiť faktory ako sú početnosť zápisu a čítania dát, počet dotazov na operácie, typ uchovávaných dát a podobne.

**Porovnanie MongoDB s ElasticSearch**

- MongoDB prichádza s kompletným balíkom riešení, v prípade ES je potrebne napríklad pre zaručenie bezpečnosti licencovať Shield v rámci Gold/Platinum programu.
- ElasticSearch má pravidlá čo sa týka indexovania, mapping môže byť vytvorený dynamicky ale jeho prípadne zmeny vyžadujú kompletnú reindexáciu všetkých dokumentov
- MongoDB v rámci svojej schémy dokáže spracovať importy rozličných dát v prípade zlých typov sa jednoducho ako slabo typované tieto dáta uložia
- Pre vyhľadávanie v ES musia byť dáta zaradené do vyhľadávacie indexu (využíva sa invertovaný index), zaručuje to široké možnosti vyhľadávania ale aj časovú potrebu pre uloženie
- GridFS v MongoDB je rozhranie pre spracovanie importov, cez rovnaké rozhranie môžeme zároveň aj čítať
- Invertovaný index pri vyhľadávaní v ES nezaručuje presný výsledok
- Možnosť vytvárania zložených kľúčov (subsetov dát) v rámci vyhľadávania pre MongoDB
- Okamžitá perzistencia dát v MongoDB

V porovnaní s ďalšími technológiami ako Redis, Cassandra opäť uvažujeme nad iným princípom dát. V prípade týchto technológií pre cache (Redis) alebo Cassandra, Hbase (forma BigData) máme jednoduchšie možnosti čo sa týka dotazovania. Najčastejšie to je pomocou kľúču, ktorý odkazuje na cieľové dáta. Redis ponúka uchovanie dát, rozličné dátové typy a rýchlu odozvu najmä pre dáta, ktoré potrebujeme na určitú dobu.

V porovnaní s BigData technológiami sú výhody Redisu najmä v tom, že dáta sú ukladané v pamäti, dokáže ukladať session data a vzťahy sú priamo key-value. Čo sa týka fault-tolerance a možností vytvárania klastrov, tu dominujú spomínané technológie Cassandra a HBase. Použitie pre ukladanie je takmer rovnaké, rozdiel týchto dvoch technológií je najmä v tom, že Cassandra má vlastnosti dostupnosti a partition tolerance, zatiaľ čo HBase miesto dostupnosti uprednostňuje konzistenciu.

**Cassandra**

- Škálovateľnost vo veľkom klástry.
- Jednoduchý prechod z RDBMS.
- Vysoký výkon pre single-row čítanie.
- Podpora Datastax (data manažment).
- Optimalizácie pre zápis.

- Nepodporuje range row-scans.
- Nepodporuje atomické porovnanie a sety.
- Neexistujú dynamické stĺpce (podpora druhoradých indexov na column-families, kde poznáme názov stĺpcu).

**Hbase**

- Silná konzistencia dát a partitioning.
- Podpora triggerov a uložených procedúr ako v RDBMS.
- Podpora Hadoop.
- Range row scans.
- Optimalizácia pre čítanie a single-write na master node.
- Podpora agregácií dát.
- Vysoký stupeň škálovateľnosti a auto-shardingu dát.
- Nevýhodou je horší jazyk pre vývoj.
- Na jeden riadok nepodporuje read load balancing.
- Operácie v riadku (inter-row) nie sú atomické.
- V prípade použitia iba jedného HBase Master node sa jedná o single point of failure.

## 2.6 Ďalšie prípady využitia

Jedným z častých prípadov je využitie technológie Redis ako cache. V prípade ukladanie používateľskej session sa uplatňuje najmä výhoda perzistencie dát. Systém vytvára v špecifických intervaloch interný snapshot dát. Jednotlivé udalosti sú ukladané do logu a z nich je možné načítať obnovu po opätovnom štarte. Okrem jednoduchých session môžu byť ukladané aj iné formy dát nakoľko Redis podporuje viacero dátových štruktúr – výhoda oproti iným cache systémom. Pre rýchly prístup sú dáta uložené v pamäti a systém sa dá použiť aj pre prácu s PUB/SUB rozhraniami (štýl fronty). Oproti iným neponúka široké možnosti spracovania dát, agregácií a perzistencie.

MemSQL ponúka najlepší výkon v prípade potreby spracovania dát v reálnom čase pričom formát dát je rôzny a samotný dataset sa mení. Dáta zo vstupu je možné transakčne spracovať komplexnými SQL dotazmi. Dodatočne je možné kombinovať real-time spracovanie s ukladaním historických dát na disk. V rámci projektov, kde sú potrebné analýzy, segmentácia dát, práca s geografickými dátami a rôzne formy business intelligence vieme využiť prvky tejto databázy

za predpokladu, že dáta je možné uložiť do pamäte RAM. Rozšírením je taktiež distribuovaná operácia spájania tabuliek, kedy je možné časť dát zaslať na ďalšie uzly, ktoré vrátia výsledok spojenia po menších častiach.

## 2.7 Porovnanie ukladania a výkonu NoSQL databázových systémov

V danej podkapitole sa zameriam na porovnanie spôsobov ukladania a práce s dátami medzi jednotlivými NoSQL databázami. Následne prevediem porovnanie možností využitia databázového systému na základe typu ukladaných dát.

### 2.7.1 Dátový model

V rámci dátového modelu budeme simulovať data v podobe údajov o autách, ktoré sú typické najmä pre inzerentné portály, ktoré ponúkajú ojazdené autá. Jednotlivé objekty budú ukladané v denormalizovanej podobe. Pre testovanie bol využitý zjednodušený dátový model, obrázok číslo 9.

### 2.7.2 Testovacie prostredie

Pre účely testovania bol zvolený virtuálny server s hardwerovým vybavením 4 jadrami CPU a 32 GB RAM s dostatočným diskovým SSD priestorom. Jednotlivé inštancie databázových systémov boli spúšťané ako virtuálne Docker kontajnere. Výhodou bola jednoduchá počiatočná konfigurácia. Jednotlivé testy – dotazy na databázu boli spúšťané asynchrónne. V rámci testu bolo uložených 100 000 záznamov vygenerovaných dát, ktoré boli podľa vyššie uvedeného modelu denormalizované.

Test bol zameraný na ukážku rýchlosti ukladania, vyhľadávania, agregovaného dotazu či aktualizácií dát. V rámci jedného testu bolo asynchrónne spúšťaných 100 dotazov s náhodnými hodnotami v rozsahu vkladaných dát. Výsledne časy zodpovedajú aplikačnému meraniu. Pri opakovanom spúšťaní dochádzalo k miernym zmenám, preto sú uvádzané priemerne časy výsledkov. Pri testoch s indexami boli po skončení odstránene, rovnako databázová cache počas behu. Časy týchto operácií nie sú zahrnuté vo výsledkoch. Vkladanie záznamov bolo atomické po dávke 250 záznamov.

#### 2.7.2.1 MongoDB

Tabuľka 1: Výsledky testovania MongoDB

| Vloženie | Vyhľadanie | Vyhľadanie * | Aktualizácia | Aktualizácia * | Agregácia |
|---|---|---|---|---|---|
| 4280ms | 2496ms | 81ms | 2868ms | 92ms | 1799ms |

Výhodou používania MongoDB je flexibilita schémy, kde môžeme v rámci kolekcie uložiť ľubovoľne formátované – normalizované dáta. Vkladanie záznamov malo spomedzi testov lepší čas. Databáza ponúka rýchle vyhľadanie a výhodou oproti relačným databázam môžu byť možnosti jednoduchšej správy a škálovania. * - *použitie indexu nad atribútom* (Tabuľka 1)

#### 2.7.2.2 Elasticsearch

Tabuľka 2: Výsledky testovania Elasticsearch

| Vloženie | Vyhľadanie | Aktualizácia | Agregácia |
|---|---|---|---|
| 51419ms | 32264ms | 3633ms | 438ms |

Výkon v rámci vkladania či práce nad dátami môže byť mierne pomalší aj v prípade viacerých použitých inštancii kvôli nutnosti reindexacie úložiska po skončení operácie. V rámci nastavení sa však dá čas, kedy sa aplikujú zmeny nastaviť. Najvyšií výkon táto dokumentovo orientovaná databáza ponúkne pri analytike či full-textovom vyhľadávaní nad dátami. (Tabuľka 2)

#### 2.7.2.3 Redis

Tabuľka 3: Výsledky testovania Redis

| Vloženie SET | Vyhľadanie SET | Vloženie HASH | Aktualizácia Hash | Vyhľadanie Hash |
|---|---|---|---|---|
| 26262ms | 5.131ms | 25804ms | 2.981ms | 2.7ms |

Oproti ostatným testovaným databázam je Redis orientovaný na ukladanie spôsobom kľúč – hodnota. V rámci výkonu ponúka výrazne rýchlejšiu prácu nad dátami, ktorá je však obmedzené použitím kľúču, ktory predstavuje "hash" hodnotu, ktorá adresuje konkrétnu hodnotu. V prípade použitia Hash Setu dochádza k úspornejšiemu ukladaniu dát. Samotný SET je vhodný pre ukladanie jednoduchých dát formou cache. Pri Hash sete môžeme pristupovať k atribútom a hodnotám, konkrétneho setu. (Tabuľka 3)

#### 2.7.2.4 Cassandra

Tabuľka 4: Výsledky testovania Cassandra

| Vloženie | Vyhľadanie | Aktualizácia | Agregácia |
|---|---|---|---|
| 16131 | 21ms | 4.2 | 6234ms * |

Ponúka vysoký výkon pri vyhľadávaní samotných dát a ich škálovaní vďaka spôsobu ukladania a práce. Dáta sú ukladané do riadkov a nad nimi je samotná tabuľka, ktorá pripadá vybranému kľúčovému priestoru, ktorý sa následne dokáže replikovať. V rámci modelu je využitie vhodné pri nízkej kardinalite dát. Vyhľadanie dát bolo prevádzané nad vytvorením primárnym kľúčom. * Agregácia dosahuje veľmi zlé výsledky. Uvedená hodnota zodpovedá jednému dotazu, dôvodom je prevádzaný "scan"nad tabuľkou. Pre analytiku nad dátami je možné Cassandru možné kombinovať s Elasticsearch či Apache Spark. (Tabuľka 4)

### 2.7.3 Použitie technológií

V rámci práce je na základe výsledkov použiť vo väčšej miere ukladanie formou cache použitím Redisu. Pre štruktúrované dáta, ktoré zahŕňajú napríklad notifikácie využijem MongoDB aj vďaka jednoduchosti používania a existujúcim rozhraniam.

# 3 Architektúra systémov pre komunikáciu pomocou správ

Spôsobov akými funguje komunikácia medzi dvoma zariadeniami alebo lepšie povedané aplikáciami, môže byť hneď niekoľko. Medzi známe riešenia patria komunikácia pomocou zápisu do súborov, zdieľaná databáza, volanie vzdialených procedúr a komunikácia pomocou správ. Práve komunikácia pomocou správ ponúka princíp, kedy získavame efektívny prenos dát so zámerom na spoľahlivosť medzi dvomi účastníkmi. Ak hovoríme o aplikáciách, na jednej strane môže aplikácia zasielať dáta, ktoré sú umiestnené na komunikačný kanál (správa uložená do fronty), odkiaľ druhá aplikácia číta tieto dáta. V prípade fungovania fronty hovoríme o aplikácií ako takzvanom spotrebiteľovi dát, nakoľko su po prečítaní odstránené z úložiska. Formát prenášaných dát nie je presne definovaný, môže predstavovať dáta vo formáte XML, JSON ale takisto aj priamo pole bajtov pričom sú v neskoršej fáze serializované. [18]

Ak by sme porovnali systém pre prácu so správami s tradičnými databázami, zistíme, že tu nie je cieľom dosiahnuť vysokú kvalitu pri perzistencií dát ale skôr zabezpečiť samotný prenos a doručenie. V prípade zlyhania sa môže celý proces prenosu opakovať. Častým prípad nastáva taktiež vtedy, keď prijímateľ správy nie je aktuálne dostupný. Takáto správa býva dočasne uložená a o doručenie sa systém pokúsi neskôr. Z uvedeného vyplýva, že môžu existovať 2 modely. V jednom z nich odosielateľ čaká na potvrdenie o prijatí a naopak po odoslaní správy už na nič podobné nečaká.

Medzi hlavné výhody patria možnosti pomerne jednoduchého spôsobu komunikácie a výmeny dát, nezávislosť priamo na implementácií jednotlivých aplikácií alebo na platforme s využitím oddelenia princípov fungovania jednotlivých strán a vopred spomenutá asynchrónna komunikácia, ktorá prínaša ďalšie benefity v podobe zvýšenia výkonu. V rámci aplikácií sme tak schopný lepšie organizovať prípadnú prácu na viacerých vláknach a odstrániť závislosť odosielateľa na prijímateľovi hlavne v zmysle čakania na výsledok. So systémom na správu jednotlivých správ môžeme pracovať ako so samostatnou jednotkou, meniť konfiguráciu, škálovať pričom jednotlivé aplikácie pracujú s vlastnými rozhraniami, ktoré definujú spôsob pre komunikáciu.

Prípadne nevýhody použitia daného systému môžu spočívať práve v asynchrónnom spracovaní dát. Systém pridáva do celkovej architektúry aplikácie mierne viac komplexnosti. Tá spočíva v náročnejších podmienkach pri hľadaní a odstraňovaní jednotlivých chýb, kedy musíme zaviesť dodatočné uchovávanie prípadných chýb a udalostí pre informačné účely a jednotlivo identifikovať prenášané správy. V niektorých prípadoch môžeme požadovať naopak synchrónnu komunikáciu. Tu nemusí byť práve zabezpečené poradie doručenia jednotlivých správ a taktiež časová doba, ktorá môže nastať v prípade spracovávania prijatých dát. Z hľadiska výkonu prenosu môže nastať určitý stupeň zníženia pri prenose, kedy sa prenáša veľa malých správ, je požadované potvrdzovanie či zabalenie a čítanie dát do správy. V neposlednom rade to je možná závislosť systému správ na konkrétnej platforme.

## 3.1 Prvky systému pre zasielanie správ

Po všeobecnom úvode u fungovaní a porovnaní vlastnosti vzoru pre zasielanie správ opíšem jednotlivé prvky, ktoré dotvárajú kompletnú architektúru celého systému. Ten je potom schopný zaručiť požadované správanie, ktoré vychádza zo špecifikácie.

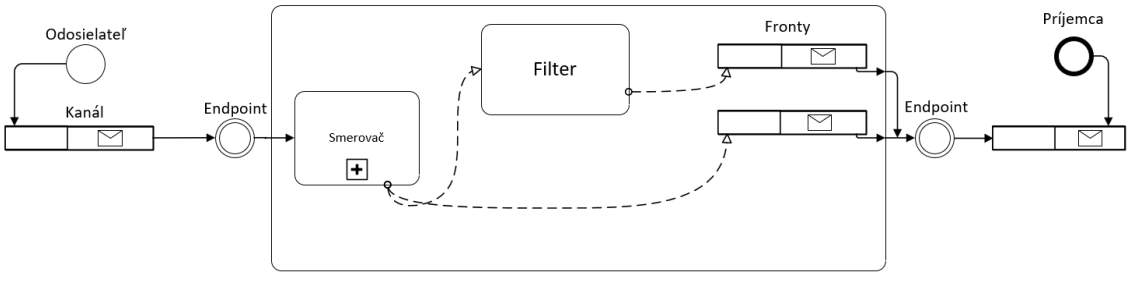
**3.1.0.1 Kanál** Predstavuje komunikačnú linku medzi odosielateľom a príjemcom. V skutočnosti sa nevytvára priame existujúce spojenie, naopak, ide skôr o abstraktný popis prenosu dát cez systém.

**3.1.0.2 Správa** Dátová jednotka, ktorá sa prenáša cez kanál medzi účastníkmi a obsahuje zapuzdrené informácie pre identifikáciu údajov o konkrétnej relácií a zasielané dáta.

**3.1.0.3 Smerovač** Časť systému, ktorá je v rámci architektúry umiestnenia do bodu, v ktorom príma správy a následne ich priradzuje do konkrétnych kanálov tak aby sa dostali k požadovaným príjemcom. Pre ukladanie môžeme považovať aj typickú frontu.

**3.1.0.4 Filtre** Z hľadiská návrhu implementované ako samostatné jednotky, ktoré nemajú závislosti na ďalšie komponenty. Využitie je práve pri transformácií dát, kedy napríklad pred samotným umiestnením do fronty, dochádza k zmene zasielaných dát. V rámci transformácie môže existovať viacero filtrov, ktoré sú prepojené abstraktnými "pipes". Zmena dát môže predstavovať napríklad prevod formátu do takého, ktorý využíva príjemca prípadne krokom, ktoré zabezpečujú autentifikáciu dát pred prijatím.

**3.1.0.5 Koncový bod** Takzvaný (endpoint) bod , na ktorý sa pripája odosielateľ a na druhej strane ďalší bod pre príjemcu. Medzi dvomi koncovými bodmi môže byť umiestnená logika celého systému pre komunikáciu medzi dvoma účastníkmi, bez toho aby vedeli o konfigurácií a ďalších interných detailoch. (Schéma 2) [19]

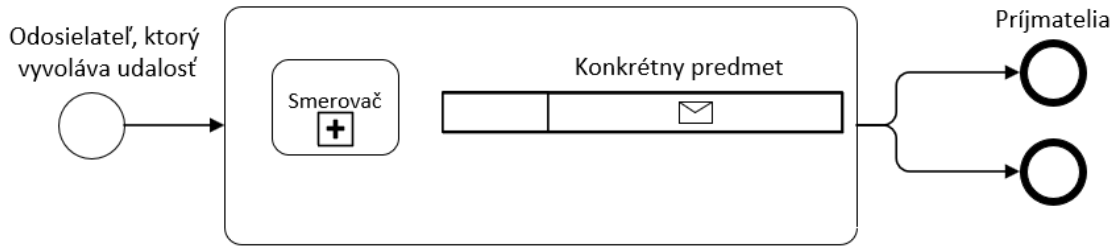



Obr. 2: Zobrazenie prvkov systému pre prenos správ

## 3.2 Vzor observer

Samotný vzor predstavuje softwarový návrh, v ktorom spracúvame vzniknuté udalosti a zasielame upozornenia – notifikácie prihláseným observerom. Na jednej strane sa nachádza predmet, ktorý obsahuje odkazy na jednotlivých prihlásených pozorovateľov daného objektu a pomocou volania obslužných metód ich notifikuje. Vzor rieši situácie, kedy máme nižšie previazanie medzi jednotlivými objektami, daný pozorovatelia závisia na odoberanom predmete, ďalej v prípade zaslania notifikácie mám zaručené doručenie pre všetkých. (Schéma 3)

Z hľadiska fungovania zasielania správ vieme tento vzor efektívne implementovať v prípade použitia kanálu PUB/SUB, ktorý je detailne popísanie neskôr. V rámci architektúry využívame existujúci kanál avšak zaslaná správa odosielateľom je skopírovaná do ďalších virtuálnych kanálov, ktoré patria jednotlivým príjemcom. Správa je považovaná za odoslanú až v prípade, že je doručená všetkým, ktorý odoberajú daný predmet ("subject"). Ďalšou podmienkou, ktorú by mal systém splňovať je tá, že každá správa pre daného príjemcu je doručená práve jeden krát a nie opakovane.




Obr. 3: Prenos správy na základe vyvolanéj udalosti

## 3.3 Smerovanie správ

Podstatnú časť fungovania prenosu správ predstavuje samotné smerovanie, ktoré môže mať vplyv na fungovanie ďalších častí architektúry a taktiež výkon. Z hľadiska smerovača hovoríme taktiež o úlohe sprostredkovateľa (brokera). Jednotlivé spôsoby smerovania môžeme rozložiť na jednoduchšie a komplexnejšie riešenia. V zásade, v každom prípade sa snažíme o spoľahlivé určenie adresy doručenia pre každú správu, ktorá prechádza danou komponentou systému. Smerovač nerieši situáciu, kedy sa nepodarí doručiť správu prípade zlyhá iný krok po určení prijímateľa prípadne ďalšieho kroku interného spracovania.

Najjednoduchší spôsob smerovania spočíva v načítaní dát, ktoré obsahuje aktuálna správa a odoslaní do kanálu, pre ktorý sú splnené podmienky na základe obsahu. Smerovač tak plní funkcionalitu, kedy odosielateľ nemusí priamo vedieť adresu prijímateľa. Daný spôsob je jednoduché použiť aj pri metóde publish – subscribe, kedy na základe tela správy určíme cieľový predmet. Pre účely smerovania môžeme využiť ďalšie prvky pre kompozíciu alebo dekompozíciu dát.

**3.3.0.1 Splitter** Umožňuje rozdeliť správu na menšie časti, ktoré sú následne spracované individuálne. Nevýhodou môže byť neskoršia potreba spätnej kompozície pôvodnej správy z menšej časti, napríklad v prípade chyby.

**3.3.0.2 Aggregator** Funguje opačne ako spomínaný splitter a to tak, že viacero správ, ktoré príjme dokáže zaobaliť do jednej a tú odoslať príjemcovi. Dokáže vnútorne udržiavať stav uložených správ, pred tím ako ich pošle ďalej.

**3.3.0.3 Resequencer** Spracováva poradie správ napríklad v prípade, že chybnú správu treba zaradiť späť v požadovanom poradí tak aby následne príjemca dostal správne dáta. [20]

## 3.4 Konfigurácia smerovania

Ako už bolo načrtnuté v prehľade jednotlivých prvkov, uvažujeme pri spracovaní aj o stave daných dát. Ten sa môže počas spracovávania zmeniť. V zásade pri štandardom postupe sa nemusíme zaoberať stavom nakoľko je ovplyvňovaní napríklad filtrami avšak naopak pri udržaní informácie stavu správy vieme identifikovať prípadné duplicity, ktoré by sa mohli objaviť v rovnaký čas. Počet krokov môže byť v rôznych fázach smerovania rozličný a preto smerovanie konkrétnej správy, môže závisieť na jej obsahu a biznis logike danej aplikácie. V prípade smerovania na základe obsahu využívame komponenty filtrov spolu s filtrovacími kanálmi, ktoré ovplyvňujú správanie konkrétnej správy avšak nie samotného smerovača, ktorý je nastavený na to aby využíval existujúce zadefinované pravidlá.

Pre dynamické spracovanie vieme konfigurovať smerovač či už pri samotnom štarte alebo behu. Dynamicky smerovať využíva pravidlá, ktoré príma od konkrétneho príjemcu a vyžaduje extra kanál, cez ktorý sa prenášajú jednotlivé pravidlá. Fungovanie je následne na základe pravidiel a smerovač odosiela na prvý kanál, ktorý vyhovuje podmienkam. Podobne ako je tomu pri pravidlá takisto aj zoznam prijímateľov môže byť dynamický a ich správanie je nezávislé. Zoznám prijímateľov zaručuje, že pre každého jedného účastníka existuje samostatný kanál cez ktorý sa komunikuje.

Transakčná úroveň môže byť rozdelená do niekoľkých stavov. Jedna transakcia predstavuje úspešne odoslanie väčšieho počtu správ a tak je úspešná až vtedy, keď smerovať umiestni všetky správy na výstupný kanál. V ďalšom prípade môžeme udržiavať zoznam prijímateľov, ktorým sme správu odoslali a napríklad v prípade chyby opakovať doručenie pričom nezabezpečujeme, že nejaká jedna správa nebude doručená dva krát. V tomto prípade chceme zaručiť jednotný stav dát.

Vopred spomenutý doručovateľ správ (broker) je opakom jednoduchej zbernice správ, ktorá existuje medzi dvoma účastníkmi a preberá prijaté dáta na vstupe a bez zložitej operácie smerovania posiela správu na druhú stranu. Doručovateľ môže prijímať od viacerých zdrojov a využitím smerovanie, ktoré bolo popísané rozhodnúť kde bude správa ďalej odoslaná.

#### 3.4.0.1 Porovnanie filtrovania správ pri smerovaní

V prvom uvedenom prípade, kedy čítame obsah dát a podľa nich určujeme cieľovú destináciu, vieme využiť pri frontách a taktiež zabezpečiť aj transakčné spracovanie pričom cieľový prijímatelia musia byť obsiahnutý v zoznamoch tak aby sme im vedeli smerovať správu. Dané riešenie je dostatočne spoľahlivé a môžeme tu použiť aj dynamické smerovanie. Naopak pri PUB/SUB nemusíme vedieť kto je priamo prijímateľom nakoľko je správa zaradená do konkrétnych predmetov a výhody tohto prístupu sú určené pre jednoduchšie dáta ako napríklad aj spracovávané webové notifikácie.

## 3.5 Transformácia správ

Ďalší popisovaný proces ma často veľký význam z hľadiska integrácie, kedy medzi sebou komunikuje viacero systémov. Avšak aj v prípade implementácie doručovateľa či koncového bodu pre správy môžeme na základe zvolených podmienok meniť samotné dáta, ktoré sú vopred odoslané prvotným odosielateľom a určené pre jedného či skupinu prijímateľov. V rámci transformácie môžeme prevádzať niekoľko podprocesov, ktoré môžu viesť k zmene dát.

Prvým spomenutým je preklad správy z hľadiska formátu. Samotné dáta môžu mať štandardizovaný ale taktiež aj vlastný formát, ktorý vychádza zo špecifikácie aplikácie. Príkladom štandardu môžu byť SOAP správy, pri ktorých definujeme obálku dát, ktorá pozostáva z hlavičky a tela dát. Každý kto prevádza čítanie nad dátami najskôr "vybalí"dáta, prečíta telo správy a v prípade zasielania zabalí správu a odošle. V prípade vlastnej štruktúry tiež využijeme metadáta, uložené v hlavičke, ktoré sa prenášajú naprieč systémom, nie sú nejak zmazané pri prenose a slúžia pre účely identifikácie aktuálneho prenosu a naopak telo správy, ktoré môže prejsť procesom transformácie.

Pri zmene obsahu môžeme využiť prvky filtrovania obsahu, ktorý však nerieši nejakým spôsobom autorizácií v rámci systému ale skôr jednoduchým spôsobom filtruje dáta, ktoré sú potrebné pre prijímateľa bez toho aby sa prenášali veľké objemy dát. Podobne funguje aj kontrola obsahu dát, ktoré by mali byť pre prijímateľa obsiahnuté. Proces funguje podobne ako filter avšak prijaté dáta, ktoré najprv odstráni zo správy uloží pre neskoršie spracovanie. Tie môže ďalej spracovať komponenta, ktorá doplňuje obsah do správ v prípade, že sú vyžadované cieľovým formátom. V rámci zaručenia fungovania integrácie medzi jednotlivými časťami taktiež využívame vopred načrtnutú normalizáciu, kedy môže nastať prípad, že odosielateľ príma dáta vo formáte XML ale prijímateľ spracováva JSON formát. Ak sa pozrieme na konkrétnejšiu biznis logiku, nájdeme tu existujúce prípady, kedy jeden systém rieši fakturáciu financií zatiaľ čo ďalší systém, ktorý prevezme prijímajúce dáta generuje konkrétne faktúry a z hľadiska fungovania tieto dva systému pracujú s mierne odlišnými dátami.

Z hľadiska architektúry softvérových aplikácií nám uvedené procesy načrtajú vzor adaptéru. Ten práve rieši nekompatibilitu medzi dvomi rozhraniami. Pre jednotlivé triedy, ktoré tak chcú komunikovať musia spĺňať potrebný kontrakt avšak ich existujúca funkcionalita nemusí byť priamo zmenená ale skôr rozšírená vytvorením dekorátoru existujúcej triedy. Pri implementácií

vytvárame novú adapter triedu, ktorá splňuje požadovaný kontrakt cieľovou triedou a taktiež dokáže prijať na vstupe objekt s pôvodnou funkcionalitou a tú využiť. [21]

## 3.6 Koncové body

Pri návrhu architektúry systému je jedným už spomínaným dôležitým kritériom nízka previazanosť jednotlivých častí, tak aby dokázali fungovať samostatne a prípadne zmeny neovplyvnili fungovanie iných komponent. Koncové body predstavujú abstraktnú bránu, ktorá oddeľuje odosielateľa a prijímateľa od samostatného systému pre prenos správ. Najčastejšia implementácia predstavuje API rozhranie, ktoré poskytuje klientom metódy pre štandardné operácie nad správami a samotné rozhranie ďalej rieši implementáciu napríklad ako komunikovať na úrovni so systémom pre správy. Zjednodušenou formou API je práve komunikačná brána, ktorá v rámci svojho rozhrania poskytuje práve základné operácie bez možnosti dodatočnej konfigurácie, ktorú už zahrňuje vo svojej špecifikácií.

Aby sme splnili nízke previazanie priamo v aplikácií či už na strane prijímateľa alebo odosielateľa, je potrebné oddeliť biznis logiku, samotnú doménu od časti infraštruktúry, kde operuje naše rozhranie. V rámci domény pracujeme s existujúcimi entitami, ktoré definujú model aplikácie. Samotná entita však nemusí predstavovať totožný objekt požadovaný rozhraním a preto pristupujeme k mapovaniu zdrojových entít na cieľové objekty a naopak. Odstraňujeme tak pripadne závislosti na iných objektoch, ktoré nemusia byť potrebné. Jednoduchým spôsobom ako vytvoriť požiadavku na rozhranie je vytvorením "Commandöbjektu, ktorý obsahuje namapované dáta a plní funkciu parametru pre obslužnú metódu konkrétnej akcie, napríklad zaslania správy. Oproti časti systému, ktorá sa stará o transformáciu dát sa mapovanie odlišuje tým, že pracuje vnútri na aplikačnej vrstve, využíva samotnú logiku a odľahčuje tak zmenu biznis dát v neskoršom kroku.

## 3.7 Vzory pre príjem a spracovanie koncovým bodom

Ak si po predošlom spojení koncového bodu priamo s klientskou aplikáciou predstavíme túto časť ako samostatný prvok, ktorý je v úlohe odberateľa správ a nezaoberáme sa priamo doručením samotnému klientovi, pripadá tak niekoľko možností ako a kedy spracúva správy daná komponenta. Uvedené vzory riešia hlavne situácie, ktoré rozhodujú o tom kedy sa uskutoční prenos správy, ako bude uložená v prípade straty spojenia s prijímateľom, či bude výsledok prenosu potvrdený, ktoré časť dát bude výsledkom operácie a ako bude prebiehať preberanie správ zo systému ak bude súčasne spustených viacero nezávislých koncových bodov.

Priebeh komunikácie môže predstavovať perzistenciu stavu, ktorá trvá pokiaľ nenastane potvrdenie z oboch strán. V jednom prípade môžeme využiť transakčný level, ktorý bol popísaný a nechať koncový bod potvrdiť prenos správy po tom ako budú dáta doručené koncovým klientom. Odosielateľ potvrdzuje odoslanie taktiež aby sa správa mohla uložiť do fronty. V momente keď nastane potvrdenie koncovým bodom, dochádza k jej zmazaniu. Podobne môžeme uvažovať pri

využití 2 abstraktných kanálov určených pre kontrolu požiadavku a odpovede v rámci jedného spojenia. Oproti transakcií prebieha potvrdenie medzi odosielateľom a prijímateľom a nie konkrétnou frontou, ktorú však pre tento účel môžu využiť ale je viazaná na spoločný komunikačný kanál.

Formu spracovanie konkrétnej jednej správy môžeme rozdeliť na synchrónnu a asynchrónnu. Synchrónne spracovanie predstavuje pre prijímateľa, ktorý odoberá správy, spracovanie na jednom vlákne. Uvedené vlákno sa okrem komunikácie stará aj o prípadný proces transformácie dáta, filtrovania a ďalší procesov a až po ich skončení dochádza k prečítaniu ďalšej správy. Asynchrónne spracovanie vyhradzuje vlákno pre prijatie správ – komunikáciu so systémom a pre spracovanie spúšťa dodatočné vlákna prípadne predáva správu samostatným aplikáciám pre spracovanie. Kedy príjme koncový bod správu ? Tu prichádza niekoľko možností. Jednou z nich je ponechanie rozhodovania na aplikačnej časti, ktorá napríklad v pravidelných intervaloch kontroluje či je dostupná nejaká správa. Výhodou môže byť nižšia záťaž na systém, prípadne hromadné spracovanie na základe modelu dát avšak nevýhodou bude rýchlosť spracovania aj v prípade, že je súčasne spustených niekoľko koncových systémov. Efektívnym spôsobom kedy prijať správu je moment, keď je doručená do cieľovej fronty. Tento spôsob je založený na udalosti, teda vložení a dochádza tak k okamžitému spracovaniu. Vhodným použitím je asynchrónne spracovanie, kedy môžeme ihneď pokračovať v komunikácií. Opačným spôsobom je vzor nazvaný ako aktiváciu služby. Tu môžeme využiť transakčný prenos či model pre požiadavku – odpoveď pretože požiadavku na komunikáciu prichádza od samotného klienta a koncový bod na základe toho kontroluje dostupnosť novej správy.

Pripojenie do systému správ môže súčasne využívať viacero koncových bodov. Tie môžu "súperiťö to, kto prvý preberie správu z fronty, pričom dosahujeme rýchle spracovanie a škálovaním služby ho dokážeme zvyšovať za predpokladu, že nepresiahneme limity systému správ. Tento spôsob vidíme pre typické modely PUB/SUB. V opačnom prípade pre každého odberateľa / bod vytvoríme samostatnú frontu a zodpovednosť, komu sa odošle aktuálna správ, prenecháme na komponentu smerovača prípade priamo odosielateľa. Výhodou je možnosť konfigurácie samotných predmetov podľa odberateľov naopak nevýhodou nemožnosť škálovať koncové body, ktoré sú viazané na konkrétne fronty. V prípade vyčerpania správ v danej fronte musí bod čakať a nepreberá správy z iných front. Dodatočné úlohy, ktoré môže riešiť koncový bod sú možnosti rozhodnúť o tom ako bude správa doručená prijímateľovi v prípade, že je odpojený. Správa môže ale nemusí byť potvrdená, zostáva vo fronte, prípade dochádza k uloženiu u samotného odberateľa. Dáta po vybratí z fronty, môžu prejsť selekciou, filtrami a využitím zadefinovanej aplikačnej logiky tak nemusia byť ďalej doručené. V neposlednom rade môžu nastať situáciu, kedy dostane odberateľ za sebou niekoľko rovnakých správ za sebou. Podobne ako v predošlom kroku, takéto nadbytočné správy môžu byť odfiltrované.

## 3.8 Monitorovanie a zaznamenávanie chýb pri prenose

Pri vytvorení systému založeného na distribuovanej architektúre jednotlivých komponentov a asynchrónnom spracovaní môžu vznikať rôzne typy udalosti, ktoré v jednom prípade vieme spracovať a očakávame ich. Najčastejšie práve rôzne upozornenia, ktoré nemusia nutne znamenať chybu, ktorá by závažne ovplyvnila celý prenos alebo na druhej strane neočakávané chyby. V prípade, že sa odoslaná správa spracúva asynchrónne a odosielateľ neskôr nedostáva výsledok spracovania musíme kvôli lepšiemu prehľadu nad samotným systémom zaviesť ďalšiu vrstvu, ktorá bude monitorovať priebeh a umožní nám dohľadať v záznamoch prebieha prenosu pre jednotlivé správy. Pre lepšiu predstavu, uvažujme o systéme, ktorý prenáša správy a nad dátami dodatočne prevádza typické kroky ako smerovanie pre doručovateľov, fronty prípadne zmeny – transformáciu v rámci tela dát. Systém môže obsahovať n bežiacich inštancií, ktorých počet a stav sa môže dynamicky v reálnom čase meniť a potrebujeme na to reagovať prípadne na základe výsledkov, ktoré vyvodíme zo záznamov z monitorovania, upravíme konfiguráciu. Pre dané účely vytvoríme centrálnu jednotku s úložiskom pre dáta, ktoré zasielajú jednotlivé inštancie. V tomto bode môžeme obsiahnuť dodatočnú konfiguráciu, ktorá poslúži aj pre novo spustená inštancie, aktuálnu kontrolu stavu, kedy jednotlivé komponenty odosielajú kontrolnú požiadavku na monitoring s informáciou o internom stave, spracovanie testovacích dát, ktoré môžu predstavovať náhodne vytvorené správy pre tieto účely a v prípade úspešného spracovania sú zahodené, ukladanie štatistických údajov napríklad o počte prenesených správ za sekundu prípadne dostupného fyzického miesta v úložisku a v neposlednom rade záznam chýb, ktoré nedokáže systém nijakým spôsobom preskočiť a pokračovať ďalej.

Aby sme dokázali v správny čas zachytiť dáta, ktoré potrebujeme, môžeme využiť niekoľko spôsobov ako monitorovať prenos ale aj ladiť na úrovni vývojového prostredia, ktoré sa môže líšiť od produkčného a obsahuje kroky, ktoré nie sú v neskoršej fáze vyžadované a môžu prípadne znižovať výkon spracovania. Komponentu pre zachytávanie správ môžeme umiestniť ako už bolo naznačené centrálne. Po tom ako smerovač spracuje správu ju v jednom prípade môžeme zachytiť pričom stále pracujeme so vstupným a výstupným kanálom, ktorý ma v tom čase určenú koncovú frontu či prijímateľa a spracovať dodatočnými kontrolnými filtrami. V podobnom prípade sme schopný podobnou komponentou odoberať správy z fronty a prevádzať nad nimi kontrolu. Tu však vzniká problém, kedy odobratá správa nie je ďalej doručená a tak musíme zaručiť jej opätovne umiestnenie na výstupný kanál aby sa mohla odoslať ďalej. V spomenutých prípadoch nepočítame s tím, že by sme prenášané dáta ukladali. Pre potreby tvorby histórie musíme jednotlivé komponenty konfigurovať tak aby každú prenášanú správu dokázali uložiť. Jedným spôsobom je použitie komponenty pre zachytenie správ prípadne odosielanie každej správy dva krát, pričom dodatočné odoslanie bude nasmerovane do fronty, ktorá je určená pre monitorovanie. V rámci jednej správy taktiež môžeme upravovať hlavičku, do ktorej zapisujeme históriu, ktorá obsahuje predošle kroky napríklad aj od iných zdrojových podsystémov. Telo dát v tomto kroku nemeníme nakoľko to nie je účelom pri tomto kroku. Každá jedna správa môže

obsahovať v hlavičke unikátny identifikátor, ktorý ju pomôže neskôr identifikovať. [22]

Komponenta pre zachytávanie správ môže fungovať aj v režime sprostredkovateľa "proxy", kedy jednotlivé správy priamo nezahadzuje alebo len jednoducho nezasiela ďalej ale dokáže komunikovať na stranu odosielateľa a taktiež aj prijímateľa. Táto jednotka obsahuje dodatočnú logiku ako má fungovať a spracúva prenášané požiadavky a odpovede spolu s adresami a údajmi o kanáloch, na ktorých boli správe posielané. Pri testovaní taktiež často dokážeme využiť testovacie správy, ktoré sú vopred definované spolu s nakonfigurovaným odosielateľom a prijímateľom a na základe testovacích dát v prípade prijatá rozhodujeme podľa podmienok testu, či boli splnené kritéria. Jednotlivé testy môžu byť spúšťane pravidelne prípadne pri zmenách v systéme. V pravidelných intervaloch kontrolujeme taktiež dostupnosť jednotlivých inštancii, kedy očakávame odpoveď a v prípade vypršania určeného časového limit pre odpoveď prevádzame kroky ako napríklad presmerovanie odosielania správ pre odosielateľov na inú inštanciu.

# 4 Webové push notifikácie

Za posledných niekoľko rokov sa zmenili pomery, ktoré určovali akým spôsobom sú oslovovaný zákazníci na internete od vydavateľov obsahu či predajcov. S postupným rastom technológií, počtu zariadení a samotných užívateľov prišli nove spôsoby ako ich osloviť a reagovať na ich dopyt. Nasledujúca kapitola sa preto bude venovať konkrétnemu prvku, ktorý je využívaný populárne nielen pre mobilné zariadenia ale dnes taktiež aj pre používateľov – návštevníkov webu a tým sú notifikácie, ktoré pridávajú možnosti pre personalizáciu a okamžité oslovenie napríklad zákazníkov. Medzí výhody patria vyššia miera záujmu, zameranie na konkrétnych používateľov a odoslanie v reálnom čase.

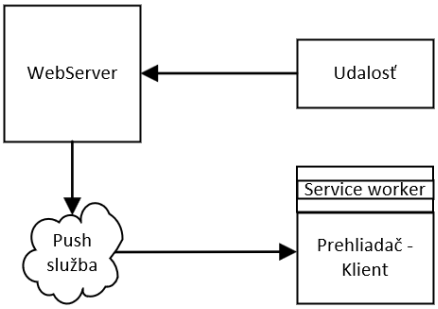
Medzi základné vlastnosti webových push notifikácií patria správy v dĺžke okolo 50 znakov, ktoré sa zobrazujú okamžite používateľovi na obrazovke, prípadne v momente keď je dostupný. Fungujú ako jednoduchý odkaz, ktorý presmeruje na predurčenú webovú adresu pričom môžu obsahovať aj dodatočný grafický obsah ale zároveň sú jednoduchšie a nepôsobia ako spam či neobsahujú iný škodlivý softvér ako tomu môže dochádzať pri niektorých formách emailových kampaní.

Prvé notifikácie sa začali objavovať v roku 2009, kedy s nimi prišla spoločnosť Apple a vývojári tak mohli odosielať „push“ notifikácie na koncové zariadenia. O rok neskôr spustil službu Google Cloud To Device Messaging aj Google a v roku 2013 boli pridané aj takzvané „rich“ prvky, ktoré rozširovali možnosti štýlovania notifikácií. Rozšíreniu pomohlo aj posun v rámci webových štandardov a vývoju medzi prehliadačmi. Medzi oblasti, ktoré využívajú danú technológiu patria ecommerece, zdravotníctvo, prostredie bánk či cestovanie a ďalšie formy online biznisu. [23]

## 4.1 Architektúra

V rámci push notifikácií nemusíme uvažovať len o webových možnostiach ale pozrieť sa aj na prvky, ktoré sa objavujú v iných technológiách – protokoloch. Takýmto príklad môžu byť aj tradičné messaging – chat aplikácie, ktoré ihneď po odoslaní používateľom spracúvajú správu alebo P2P programy či protokoly ako IRC alebo XMPP. Starší protokol CDF, ktorý integroval Microsoft či Netscape CDF sa vo svete webu neujal a nahradil ho RSS, ktorý sa na druhej strane pre používateľa správa ako „pull“ systém.

Web push notifikácie vznikli zo zámeru IETF v rámci protokolu HTTP 2. Prenos ja viazaný na konkrétne používateľské pripojenie, komunikácia je v reálnom čase a všetko dokáže spravovať jediný server. Samotná technológia patrí do štandardu W3C a je definovaná ako API rozhranie pre používateľský webový frontend. (Schéma 4) Serverové push notifikácie pracujú ako HTTP Stream, ktorý prebieha medzi serverom a klientom. Pre zabezpečenie komunikácie existuje viacero mechanizmov. Jedným z nich je WebSocket API dostupné v HTML 5 a komunikácia prebieha cez full-duplex TCP spojenie. Spojenie zo strany serveru zostáva otvorené a webové serveri pracujú použitím funkcie CGI (Non-parsed Header Scripts). Ďalším mechanizmom je




Obr. 4: Zobrazenie prenosu push notifikácie

špeciálny MIME typ multipart/x-mixed-replace vyvinutý v roku 1995 spoločnosťou Netscape. Predstavuje princíp meniaceho sa dokumentu pokiaľ potrebuje server odoslať novú požiadavku. Mechanizmus WHATWG Web Applications zahrňuje podobný princíp. Bol použitý ako experimentálna vlastnosť v prehliadači Opera nazvaná Server-Sent-Events a dnes je štandardom v rámci HTML 5. [25]

## 4.2 Dáta a spracovanie chýb

Konkrétne v uvádzanom modelovom príklade je potrebné aby sme ukladali tokeny priradené k identifikátorom používateľov a následne samotné dáta pre správy, ktoré budeme zasielať. Identifikácia používateľa nám pomôže zamedziť opakovanému zasielaniu rovnakej správy, kontrolovať stav odoslania, využívať neskôr dáta pre účely vytvorenie štatistík, sledovať aktivitu používateľov, takzvanú retenciu v prípade, že kliknú na odkaz, ktorý obsahuje notifikácia. Priradený token následne používame pri samotnej komunikácií pričom počítame, že sa môže zmeniť v prípade odhlásenia používateľa z odberu, zmeny jeho nastavení prípadne novej registrácie. Používateľský identifikátor zostáva unikátny a nemení sa v priebehu životnosti aplikácie.

Pre konkrétneho používateľa využívame parameter "to", ktorý obsahuje jeden registračný používateľský token zatiaľčo použítim parametru "registration_ids"vieme jedným dotazom na server FCM odoslať správu väčšiemu počtu (okolo 1000 zariadení) naraz. V novších verziách sa však stretávame s parametrom "topic", ktorý predstavuje predmet definovaný ako reťazec. Konkrétny predmet môže byť odoberaný viacerými zariadeniami a taktiež pri odoslaní, ktoré podporuje aj logické operátori AND a OR, definovať príjemcov.

Pri využití protokolu HTTP spracúvame kódy z odpovedí server po odoslaní našej správy. Pri sťahovaní, kedy sa klient pripája na FCM server aby prebral správu priradzuje služba aj dodatočné chybové hlášky, pričom kód 200 má charakteristiku statusu aktuálne zasielaného požiadavku. V nasledujúce tabuľke sú zobrazené časte chybové status pri komunikácií. Ďalšiu chyby môžu byť spôsobené nesprávnym formátom dát prípade prekročením hodnôt napríklad v prípade TTL alebo veľkosti správy, kde je štandard 4096 bytov pre správu a konkrétny predmet 2048.

Tabuľka 5: Stavové kódy pre Google FCM

| Kód | Popis |
|---|---|
| 200 | Úspešne doručenie |
| 400 | Nesprávny formát odosielanej správy |
| 401 | Problém s autentifikáciou |
| 500 – 599 | Chyba serveru |
| TopicsMessageRateExceeded | Prekročenie limitu pre odberanie konkrétneho predmetu |
| DeviceMessageRateExceeded | Prekročenie limitu pre príslušné zariadenie |
| error:MismatchSenderId | Nesprávne ID (napríklad pre identifikáciu webového prehliadaču) |
| error:NotRegistered | Expirovaný token,<br>odhlásenie odberu používateľom,<br>aktualizácia aplikácie |

Pokiaľ v aplikácií nastane počas komunikácie chyba využívame v konkrétnom prípade jednoduchý spôsob ošetrenia, kedy napríklad v prípade nezaregistrovaného používateľa meníme stavový kód v databáze a naopak pri chybe v komunikácií ukladáme záznam o neúspešne odoslanej správe, ktorý môže neskôr využiť skript pre opakované odoslanie. (Tabuľka 5) [26]

## 4.3 Rošírenie pre mobilnú platformu

Pri mobilných zariadeniach máme možnosť odosielať push notifikácie pre zariadenia využívajúce operačný systém Apple iOS alebo Google Android. Možnosti, ktoré ponúka mobilná platforma zostávajú takmer rovnaké pre obe platformy, rozdiely sa týkajú konkrétnych implementácií najmä pri komunikácií. Oproti webovým notifikáciám ponúkajú tie mobilné viacero prvkov a parametrov, ktoré je možné pri zasielaní požiadavky na server nastaviť a tak upraviť podobu prijatej správy koncovým zariadením.

Pri odosielaní push notifikácií pre mobilné zariadenia máme ďalšie možnosti pre vytvorenie požiadavku. Prijatá notifikácia zohľadňuje aktuálny stav aplikácie. Môže sa tak jednať o správu, ktorá je určená priamo pre aplikáciu a tá môže vyvolávať notifikácie lokálne, prípadne pre používateľa, kedy sa zobrazuje v panely notifikácií alebo ako popup správa aj v prípade, že nie je zariadenie práve používané.

Priorita je dodatočný parameter, ktorý obsahuje číselné hodnoty (5-10) prípadne reťazce "lowä "high". Štandardne sú správy zasielanie s normálnou prioritou, kedy sa kladie dôraz hlavne na úsporu batérie zariadenia. Naopak pri vysokej priorite môžeme docieliť okamžité zobrazenie na obrazovke. V prípade neaktivity aplikácie, ktorá beží na pozadí existuje parameter "content_available"pre jej zobudenie a spracovanie dát. Pre iOS zariadenia využíva Google Firebase v tomto prípade APNs site pre doručenie a nie štandardnú FCN. Štandardná živnosť pre uloženie správy sú 4 týždne pričom sa táto hodnota dá konfigurovať aj na nižšiu hodnotu. Systém

iOS ďalej ponúka možnosť pozmeniť prijatú správu pred tím ako bude zobrazená na obrazovke od verzie 10.

Medzi rozlišujúce prvky oproti webu patria možnosti ako vyvolanie zvukového efektu, ktorý je uložený v knižnici zvukových nahrávok samotnej aplikácie, pozmenenie ikony aplikácie v zozname, dodatočné titulky, lokalizácia textu podľa konkrétnej lokácie používateľa a pre Android zariadenia aj nastavenie farby, pomocného označenia, ktoré v prípade prijatia rovnako označenej správy nahradí pôvodnú pokiaľ sa stále zobrazuje.

## 4.4 Apple APNs

Alternatívna sieť oproti Google FCM, ktorá je využívaná najmä zariadeniami značky Apple ale taktiež podporovaná pre prenos notifikácií pri použití služby Firebase prípadne inej tretej strany. Služba ponúka zabezpečený a škálovateľný prenos pre zariadenia na systémoch iOS, watchOS, tvOS a macOS. Zabezpečenie komunikácie pre účely overenia zdroju, ktorý príma a odosiela správe je možné realizovať pomocou spojenia založeného na bezpečnostných tokenoch, kedy odosielateľ správy podpisuje dáta privátnym kľúčom, prípadne využitím bezpečnostných certifikátov, kedy je však potrebné zaručiť nainštalovanú certifikačnú autoritu, konkrétne koreňový certifikát na koncovom zariadení. Sieť ďalej rieši prípady, kedy sa zasiela pre rozlične typy zariadení a odosielateľ aj v tomto prípade pracuje s unikátnym tokenom, ktorý identifikuje zariadenie, dočasné uloženie správy v prípade, že sa nedá doručiť – zariadenie neprijíma dáta a zjednotenie viacerých dát do jednej notifikácie. Medzi prvé systémy, ktoré podporovali notifikácie patria iOS 5.0, Mac OS X 10.7 pričom novšia verzia 10.9 akceptuje aj notifikácie z webových zdrojov priamo v zariadení. [27]

## 4.5 Analýza spracovania notifikácií pre webové aplikácie

V rámci vytvorenia implementácie programového rozhrania sa zameriam na analýzu možnosti zasielania webových push notifikácií medzi koncovým bodom, backend časťou a používateľom, konkrétne samotným prehliadačom. Výsledkom bude popis architektúry a možností pre komunikáciu pomocou push notifikácií, prehľad architektúry, podpora prehliadačov, analýza možnosti zasielania pre mobilné zariadenia a spôsob akým budú jednotlivé dáta ukladané. V rámci implementácie následných asynchrónnych udalostí vytvorím porovnanie s možnosťou alternatívnej komunikácie pomocou dodatočného protokolu okrem spomínaného Push. Samotný popis bude obsahovať informácie pre pochopenie spracovávanie udalosti na pozadí v rámci prehliadaču a asynchrónnu komunikáciu pomocou rôznych protokolov.

Samotný štandard pre zasielanie push notifikácií vychádza z dokumentov W3C. Push služba predstavuje bod, cez ktorý dokáže aplikačný server zasielať správy samotnému klientovy nezávisle na tom či je aktuálne webová aplikácia, strana používateľa, aktívna alebo nie. Služba sa snaží o spoľahlivé doručenie a maximálnu efektivitu. Na pozadí používateľa sa nachádza Service

Worker, ktorý sa stará o samotnú komunikáciu a správu lokálneho stavu. Celá komunikácia prebieha asynchrónne a nedochádza tak k blokovaniu na strane používateľa.

### 4.5.1 Popis Push protokolu pre webové notifikácie

Na jednej strane sa nachádza klient prezentovaný webovým prehliadačom a na druhej aplikačný server, ktorý využíva push službu pre komunikáciu. Správa reprezentuje samotné zasielané dáta a môže byť doručená aj v prípade, že klient nie je momentálne aktívny v rámci webovej aplikácie. Samotné zasielanie je možné po nadviazaní odberu – spojenia, kedy dochádza k registrácií služby v rámci server workera. Medzi dostupné využívané rozhrania patria Google Firebase a Apple PN Service.

Registrácia zahŕňa odkaz na cieľový bod, ktorý predstavuje absolútnu URL adresu, na ktorú sa zasielajú správy, čas exspirácie, kedy sa invaliduje odber, šifrovaný kľúč pre zabezpečenie komunikácie tak aby komunikácia prebiehala len medzi daným klientom a koncovým bodom. Pri vytvorení, zmazaní či iných zmenách odberu dochádza k volaniu definovaných udalostí.

Na strane klienta taktiež dochádza v prvom rade k získaniu oprávnenia pre používanie rozhrania. Samotný používateľ ho môže alebo nemusí povoliť. Po následnom povolení sa vygeneruje kľúč pre daný server a pomocou PushSubscription sa odošle. Aplikačná časť nekomunikuje priamo s prehliadačom ale cez push službu, ktorá spracuje požiadavku, overuje komunikáciu a doručuje správu. Takúto službu predstavuje napríklad Google Firebase Cloud messaging, ktorý je štandardne podporovaný v dostupných prehliadačoch. Komunikácia funguje v rámci šifrovaného spojenia. Správa je doručená následne v momente, keď je zariadenie online.

### 4.5.2 Obecný popis Publish – Subscribe prenosu správ

Štandardné fungovanie preposielania správ môžeme aplikovať aj mimo tradičné spracovanie medzi webovým klientom a serverom. Správa predstavuje dáta, ktoré sa prenášajú od zdrojového odosielateľa, následne je uložená a na konci prečítaná prijímateľom. O zaslanie správy sa starajú jednotlivý vydavatelia, ktorý ukladajú správu do fronty, ktorá môže byť rôzne kategorizovaná podľa typu správy a určená pre konkrétnych príjemcov. Následne táto skupina príjemcov dostáva správu, ktorá je pre nich určená. Samotný vydavateľ nekomunikuje priamo s príjemcom. V rámci prenosu správy má systém možnosť flexibilne pracovať so samotnými dátami. Správa je najčastejšie vydavateľom, zdroj, ktorý publikuje zdrojové správy, doručená do centrálneho systému na spracovanie. Tu môže dochádzať k transformácií správy, napríklad ak je prenášaná medzi viacerými systémami prípadne ku kategorizácií. Kategorizácií zodpovedajú jednotlivé fronty, do ktorých je správa začlenená. [28]

### 4.5.3 Fronty správ a ich spracovanie

Podobne označované ako fronty pre hromadné spracovanie úloh. V danom kontexte hovoríme o spracovaní formou FIFO. Sú využívané pri spracovaní, kedy jednotlivé uložené úlohy obsahujú

dáta pre koncové systémy, tzv. “workers”, ktorý distribuované spracúvajú prijímané úlohy z fronty. Výhodou tohto spracovania je možnosť nastaviť jednotlivým úlohám – správam prioritu, začleniť ich do kategorizovanej fronty, dosahovať optimálny čas pre spracovanie sady úloh, nakoľko vieme určiť priemerný čas spracovanie jednej úlohy a takto neustále využívať systémové zdroje. Samotné systémy môžu pracovať distribuovane, vieme dosiahnuť optimalizáciu zdrojov, plánovanie časovo náročných úloh a nižšiu závislosť medzi systémami.

Nižšia závislosť medzi systémami predstavuje opak fungovania architektúry klient – server, kedy obidve strany navzájom komunikujú a musia o sebe vedieť. Pri PUB/SUB vieme docieliť, že jednotlivý vydavatelia ale aj príjemcovia správ pracujú samostatné a navzájom o sebe ako interne fungujú nemusia vedieť. Pri podobnej architektúre vieme službu aj vďaka nízkym závislostiam škálovať a tak pracovať aj s tisíckami inštancií, ktoré sa starajú o publikovanie či spracovanie dát. Nevýhodou je nižšia garancia doručenia. Samotný systém môže pracovať počas určitej doby, kedy sa snaží správu doručiť opakovane ale nastavenia jednotlivých front predstavujú jednoduché predanie k príjemcovi. V prípade, že spracovanie u príjemcu nejakým spôsobom zlyhá, daná vrstva architektúry systému ako celku nerieši. Na rad tak prichádza dodatočná vrstva, ktorá sa stará o chybné / nespracované úlohy a ukladá hlásenie do záznamov – logov. Riešenie dodatočných chýb tak predstavuje komplexnejší problém. Ďalšou nevýhodou môže byť pomalé spracovanie v prípade nízkeho počtu systémov pre spracovávanie prípadne naopak ich nevyužívanie pri nízkom počte správ. Za potencionálne riziko musí byť zvážená aj bezpečnosť, kedy by sme mali zaručiť najmä doručovanie konkrétnym a správnym príjemcom čo môže byť dosiahnuté zabezpečením siete či použitím šifrovaného spojenia.

K asynchrónnemu spracovaniu tak dochádza už v prípade odoslanie správy klientom, moment kedy je publikovaná. Samotný klient môže pokračovať vo svojej práci a v prípade potreby spracovať výsledok po doručení. K smerovaniu správy pre konkrétnych príjemcov môže dochádzať už pri odoslaní, kedy je doručená konkrétnej skupine prijímateľov alebo všetkým. Ako bolo naznačené vyššie, samotná správa môže byť počas doručovania pozmenená. Pri jednoduchých modeloch ako napríklad doručenie informačných dát webovému klientovi nemusí dochádzať k prechodom medzi viacerými systémami. Na druhej strane, pri väčšej zložitosti samotnej architektúry môžeme spomenúť princípy Flow-based programovania, kedy fungovanie aplikácie, jednotlivé procesy pozostávajú z viacerých krokov, ktoré sú prevádzané sekvenčné. Procesy môžu predstavovať jednotlivé menšie systému aplikácie či malé komponenty, ktoré obsahujú aplikačné kód. Prvky pracujú nezávisle a predávajú postupne jednotlivé správy ďalej, až kým nedôjde k požadovanému výsledku.

#### 4.5.3.1 Rozdiel medzi frontou a PUB/SUB

Hlavný rozdiel, ktorý môžeme vidieť medzi uvedenými spôsobmi spracovania správ je práve pri ich čítaní. Zatiaľ čo pri čítaní z fronty je správa vybraná a prečítaná konkrétnym odberateľom, pri PUB/SUB hovoríme o multicaste, kedy zaslaná správa je prijatá najčastejšie do ëxchangeödkiaľ je odoslaná všetkým odberateľom. V terminológií fronty neriešime kategorizáciu.

### 4.5.4 Technológie a vzory pre implementáciu front a prenos správ

Z predošlej kapitoly, v ktorej som uviedol základne informácie o prenose a práci so správami, jednotlivé štandardy, sa tak môžeme na nasledujúce uvedené technológie pozerať na základe ich jednotlivých vlastností a požiadavkami, ktoré kladieme na výslednú funkcionalitu. V prvom prípade je to potreba implementácie jednoduchšej fronty pre správy, kde medzi požiadavky môžu patriť vlastnosti ako rýchlosť zaradenie do fronty, prioritizácia, kategorizácia front, škálovanie a využitie pre konkrétnu architektúru.

**4.5.4.1 AMQP** Dostupný protokol, ktorý predstavuje otvorený štandard pre prácu so správami pre aplikácie - technológie, ktoré by ho chceli implementovať. Medzi hlavne prvky štandardu patria bezpečnosť a interoperabilita medzi rôznymi systémami. Oproti existujúcemu API rozhraniu pre Javu – JMS sa rozlišuje v tom, že neponúka priame rozhranie API a definuje protokol nad samotnou transportnou vrstvou. Verzia 1.0 vydaná v roku 2011 zahrňuje definície pre enkódovanie rôznych typov formátov správ, popis fungovania nad linkovým sieťovým protokolom, kde využíva vlastná formát prenášaných rámcov a samostatný prenos, šifrovanie prenosu, mechanizmy pre prenos správ, ich filtrovanie a popis fungovania jednotlivých uzlov. Implementácia štandardu využívajú technológie ako Apache Qpid, StormMQ či RabbitMQ. Staršie verzie pred 1.0 sa zameriavali menej na definíciu samotného protokolu a viac na popis implementácie prenosu správy.

Ďalšie podobné protokoly sú STOMP, podobný protokolu HTTP, ktorý definuje jednoduchším spôsobom samotné prvky komunikácie, starší XMPP založený na XML a pôvodne vyžívaný na real-time chat komunikáciu (Jabber) a MQTT zameraný na definovanie PUB/SUB rozhrania, štandardizovaný ako ISO štandard s nízkymi nárokmi na sieťový prenos. [29]

**4.5.4.2 ZeroMQ** Pre potreby implementácie vlastnej práce so správami napríklad v prípade veľkej aplikácie, príklad pre FBP je možnosť využiť danú knižnicu, ktorá ponúka vysoký výkon v rámci asynchrónneho spracovania pre distribuované systémy. Obsahuje funkcionalitu pre prácu vo forme fronty správ ale taktiež aj agenta, ktorý sa stará o prenos správ medzi odosielateľom a prijímateľom. Spojenie je vytvárane pomocou socketov (Berkeley sockets). Samotná implementácia knižnice predstavuje nízko-úrovňový software a pre populárne programovacie jazyky existujú dodatočné implementované rozhrania. [30]

**4.5.4.3 Apache Qpid** Open-source projekt, ktorý ponúka dostupné knižnice pre C, C++ a Javu a ďalšie nástroje, ktoré sú postavené nad štandardom AMQP. Vďaka dostupným zdrojom je tak možné vytvoriť komunikáciu medzi jednotlivými aplikáciami, ktoré budú využívať vlastnosti prenosu správ. Existujúce rozhrania je možné využiť pre zasielanie/prijímanie správ, správu prenosu správ, ktorý využíva vlastnosti ako vytváranie skupín správ, TTL pre správy, autentifikáciu, limitovanie zdrojov, transakcie, správu nedoručených správ, replikáciu front a škálovanie pri ktorom je možnosť využiť nástroj Dispatch Router, ktorý dokáže pracovať nad

topológiou, sledovať zmeny, vykonávať konfiguráciu a neposlednom rade prepájať koncové uzly a smerovať dáta. [32]

**4.5.4.4 Redis PUB/SUB** Technológia využívaná najmä pre účely ukladania dát preferovaná pre účely v podobe cache úložiska ponúka implementáciu PUB/SUB od verzie 2.0. Jednoduchší spôsob fungovania avšak nezahŕňa perzistenciu dát, garantovania doručenia a dodatočné optimalizácie pre cluster. Perzistencia pre to musí byť v prípade potreby riešenia na ďalšej úrovni aplikácie po získaní správy. Pri doručovaní mechanizmus zahodí správu aj v prípade, že nie je žiaden prijímateľ, prípadne nastala u niektorého chyba. Od verzie 3.0 dostal Redis aj funkcionalitu pre kláster v podobe broadcastu pri zasielaní správy. Použitie je tak stále pre jednoduchšie účely.

Pattern-matching subscriptions pridávajú podpora odoberať správy z viacero kanálov pomocou špecifikácie globálneho kanálu, ktorý odpovedá regulárnemu výrazu. Rozhranie nemá prepojenie na úložisko (kľúče, ktoré odkazujú na uložené dáta), rozšírením je Keyspace notifications, ktoré príma udalosti vyvolané nad daným kľúčom, príkladom je príkaz, ktorý ovplyvní dáta, exspirujúci kľúč či kľúč, ktorý príma LPUSH notifikáciu od klienta a využitím spätného volania (callbacky) odosiela odpoveď.

**4.5.4.5 RabbitMQ** Aktuálne jedna z najviac komplexných technológií pre prácu s frontou a využívaním princípov publish – subscribe. Využíva priamo implementácie protokolov, najmä AMQP, ale taktiež aj MQTT a STOMP. Samotné fungovanie môže byť na princípe jednoduchej fronty, ako tomu je pri Redise avšak vďaka dostupnej konfigurácií taktiež ako pokročilý systém na prácu so správami. Jednou dostupnou konfiguráciou môže byť vytvorenie “exchange”, kedy je každá prijatá správa následne smerovaná do príslušnej fronty. Tu RabbitMQ ponúka viacero režimov ako "direct", kedy je správa uložená do fronty podľa parametru v dátach, "topic", uloženie do fronty podľa regulárneho výrazu a fanout, kedy je správa duplikovaná a uložená vo všetkých frontách.

Spoľahlivosť pri perzistencií a doručovaní môže byť zaručená fungovaním systému vo forme clusteru, potvrdzovaní prijatých správ a vysokej dostupnosti. V prípade replikácie sa kopírujú všetky dáta spolu s aktuálnym stavom front. Dodatočná podpora pluginov napríklad pridáva podporu aj pre AMQP 1.0. Tá nie je štandardne zahrnutá kvôli zvýšenej komplexite. Prenos dokáže fungovať aj pomocou protokolov ako HTTP, STOMP – Websockets, JSON-RPC. Medzi ďalšie prvky patria možnosti ako sú, viacero možností autentifikácie, šifrované spojenia, monitorovanie a "kontroly zdravia"systému, správa zdrojov, pokročilé nastavenia front a dostupné pluginy. [31]

**4.5.4.6 Apache Kafka** Projekt vedený ako open-source je využívaný najčastejšie pre funkcionalitu fungovania spracovávania streamov dát. Kafka ponúka nízku latenciu, real-time spracovanie a spomedzi dostupných riešení veľmi dobré možnosti pre škálovanie. Platforma ponúka

niekoľko API rozhraní pre zasielanie správ, odoberanie a streamovanie spolu s dodatočným rozhraním pre implementáciu spojenia medzi vyzbraným systémom a stream serverom. Dáta môžu byť uchovávané v jednotlivých streamoch nazvaných ako fronta predmetov. Uloženie môže byť aj v rámci viacerých partícií a pri čítaní/zapisovaní je zohľadnené poradie dát podľa danej operácie. Partície je možné distribuovať medzi viacerími jednotkami s možnosťou dodatočnej konfigurácie konkrétneho regiónu či datacentra.

Kafka funguje ako v režime fronty tak aj rozhraní pre posielanie-príjem správ. Zaručenie poradia prijatia správy z fronty, v prípade, že paralelne odoberá viacero prijímateľov dokáže vďaka prideleniu konkrétnej partície, na ktorej sú dáta konkrétnemu prijímateľovi ale taktiež naďalej fungovať paralelne a vyvažovať záťaž. Ukladanie dát je možné na disk s dodatočnou replikáciou a potvrdením o úspešnom zápise pre zasielateľov správy. Streamovanie dát je vhodné pre aplikácie, ktoré potrebujú nad prijatými dátami v konkrétnej fronte vykonávať dodatočné spracovanie, agregovať či spájať iné dáta s odlišných streamov dohromady. Takto spracované dáta môžu byť uložené do cieľovej fronty alebo odoslané aktívnym prijímateľom. Kontrola škálovania je použitím Apache Zookepera. [33]

**4.5.4.7 ActiveMQ** Populárne open-source riešenie často nasadzované pre potreby integrácie rôznych systémov. Podporuje štandardne protokoly pre prenos správ s dostupnými klientskymi knižnicami pre rôzne programovacie jazyky spolu s rozšírením o podporu transportných protokolov. Podobne ako pri predošlej technológií Apache Kafka tu nájdeme implementáciu skupín správ, ktoré je možné škálovať, zaručiť doručenie aj pre viacerých odberateľov, kedy definujeme skupinu ako virtuálnu a tak je možné spracovávať správy z jednej skupiny niekoľkými prijímateľmi paralelne. Pre škálovanie je podobne použitý Apache Zookeper.

Ako nadstavba pre spracovanie správ existuje ActiveMQ Apollo, ktorý sa líši modelom práce na viacerých vláknach a architektúrou odosielania. Medzi dodatočné vlastnosti patria autentifikácia, šifrovanie, REST API pre správu, výmena správ medzi diskom, v prípade, že sa nezmestia do pamäte RAM prípade z hľadiska šetrenia, odber správ v režime, kedy sa nezmažú po prečítaní, nastavením exspirácie dát, výberom fronty z hľadiska smerovania pre uloženie či spoľahlivým doručovaním. [34]

**4.5.4.8 Amazon SQS** Služba prestavuje jednoduchú implementáciu fronty určenú najmä pre komunikáciu medzi distribuovanými systémami, micro-aplikáciami a službami. V prvom prípade je možnosť konfigurácie fronty pre zasielanie-odber správ, kedy nemusí byť zaručené doručenie práve jeden krát či poradie. Výhodou je neobmedzená priepustnosť. V ďalšom prípade existuje FIFO fronta, ktorá zaručuje doručenie a taktiež aj poradie. Medzi vlastnosti SQS patria veľkosť obsahu dát o veľkosti 256Kb, hromadné zasielanie maximálne 10 správ v závislosti na veľkosti, uloženie správ po dobu 14 dní, zdieľanie medzi rôznymi Amazon účtami, šifrovanie na strane servera a manažment pre škálovanie, obnovu dát a prioritizáciu spracovania.

## 4.6 Implementácia prenosu využitím Google Firebase Cloud Messaging

V rámci implementácie sa zameriam na vytvorenie rozhrania, ktoré bude komunikovať so službou Google Firebase a spravovať vytváranie odberov, zasielanie správ a uchovávanie údajov o používateľoch a spracovaných dátach. Jednotlivé dáta budú uchované v nerelačnej databáze MongoDB a spracované skriptom vytvorenom pomocou NodeJs. Rozhranie ponúkne funkcionalitu, ktorú bude možnosť použiť v iných projektoch, ktoré budú chcieť implementovať zasielanie push notifikácií používateľom. Porovnanie bude prevedené s existujúcimi riešeniami, ktoré pôsobia ako dodatočné nástroje pre komunikáciu. Implementácia sa zameriava pre webové využitie.

Na strane klienta sa jedna o implementáciu skriptu v Javascripte, ktorý beží v rámci aplikácie v samotnom prehliadači. Pre aplikačné účely je vytvorené REST rozhranie v NodeJS, ktoré ponúka dostupné endpointy znázornené v tabuľke číslo 6 pre spracovanie používateľských registrácií pre prijímanie notifikácií.

Tabuľka 6: API endpointy pre odosielanie pomocou Google FCM

| Metóda | Endpoint | Dáta | Formát | Popis |
|---|---|---|---|---|
| GET | / | | application/json | Výpis odoberateľov |
| POST | /subscripction | user_id: , token: | application/json | Registrácia odberu |
| DELETE | /subscripction | user_id: | application/json | Zrušenie odberu |
| POST | /send | title: , body: , icon: , url: | application/json | Odoslanie notifikácie odberateľom |

Pre využitie služieb Google Firebase je potrebná registrácia a vytvorenie projektu pre danú aplikáciu. Pri projekte máme možnosť vybrať geolokáciu pre ukladanie dát. Po vytvorení sú vygenerované bezpečnostné API kľúče, projektové ID a pre Cloud Messaing atribúty ako Sender ID pre identifikáciu našej aplikácie ako odosielateľa a pre autentifikáciu Server Key spolu s Legacy Server key, ktorý je skrátenou podobou pôvodného kľúču a využívaný v novších verziách knižníc Firebase.

Klientsky skript importuje hlavné a messaging knižnice pre prácu s Firebase. V inicializačnej fáze priraďuje autentifikačné údaje a doménové údaje pre komunikáciu s koncovým serverom. Podmienkou pre využitie notifikácií je podpora samotného prehliadača. Používatelia, ktorý využívajú staré verzie prehliadačov tak nemôžu využiť komunikáciu pomocou notifikačných správ. V rámci aplikácie rozlišujeme dve strany notifikácií. Na jednej strane sú to notifikácie, ktoré sú prijímané používateľom pokiaľ sa nachádza na aktuálnej stránke a tie, ktoré sú prijímané na pozadí mimo aktívnu stránku.

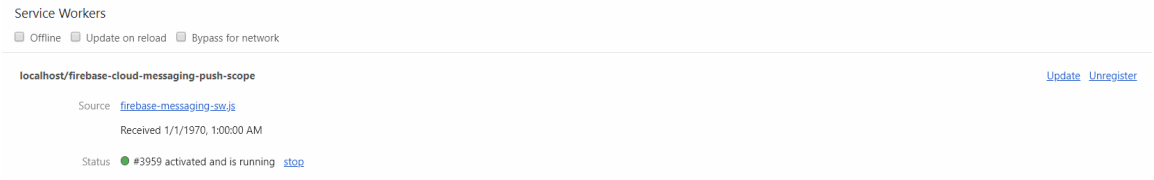
Pre aktivovanie prijímania notifikácií je potrebná používateľská interakcia v podobe povolenia oprávnenými pre prijímanie. Jednotlivé nastavenia v prehliadači môžu byť kedykoľvek nezávisle na stave aplikácie zmenené a je nutné s nimi počítať. V prípade povolenia prichádza

k vygenerovanie identifikačného tokenu, ktorý sa ukladá lokálne a taktiež na náš aplikačný server pre potreby identifikácie odberu. Jednotlivé kroky sa opakujú v prípade aktualizácie tokenu alebo v prípade jeho zmazania. Po registrácií tokenu dochádza k vyvolaniu udalosti doručenia správy v prípade prijatia, ktorá obsahuje obslužnú funkcionalitu. Medzi vlastnosti notifikácie, ktorá sa zobrazuje ako vyskakovanie okno v rohu prehliadača patria nadpis, obsah, ikona a odkaz, na ktorý je používateľ presmerovaný po kliknutí.

```
const notification = new Notification("nadpis", { icon:"http://example.example/
    icon.jpg", body: "obsah notifikacie"});
notification.onclick = function () {window.open("http://www.vsb.cz");};
```

Výpis 7: Vytvoreniu notifikácie v prehliadači

Fungovanie na pozadí zabezpečuje Service Worker, ktorý oproti skriptu pre prijímanie notifikačných dát nevyužíva časť "dataäle "notification". O jeho registráciu sa stará priamo Messaging knižnica a taktiež o zobrazovanie vyskakovaniach správ. (Schéma 5)




Obr. 5: Zaregistrovaný service worker v prehliadači

### 4.6.1 Rozšírenie funkcionality v Google Firebase Cloud Messaging

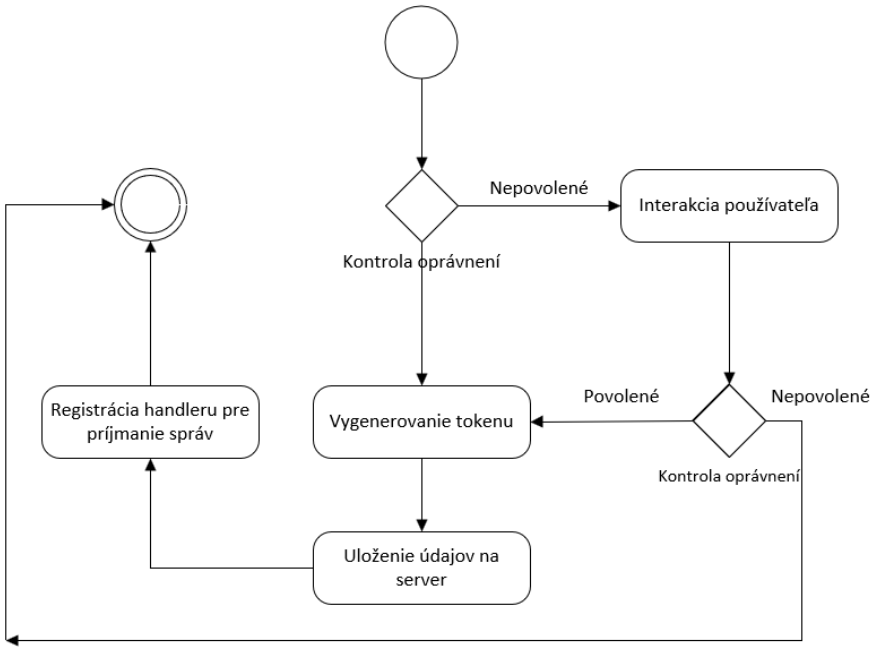
Pri odosielaní na aplikačnom serveri, máme možnosť špecifikovať niekoľko parametrov. Samotná dokumentácia obsahuje kompletný popis spolu s odpoveďami zo strany serveru.

Tabuľka 7: Parametre požiadavku pre Google FCM

| Atribút | Popis |
|---|---|
| to | Token príjmateľa správy |
| registration_ids | Pole prijímateľov, max 1000 (multicast) |
| priority | Priorita pre odoslanie |
| time_to_live | Čas uchovania notifikácie pre doručenie, štandardne 4 týždne |
| Payload: Notification | Atribúty nadpis, popis, ikona, url odkaz pre web. |
| Payload: Data | Dostupné ľubovoľné dáta |

Medzi dostupné protokoly pre komunikáciu patria XMPP, HTTP protokol a novší HTTP API zdroj. Jednotlivé protokoly sa líšia zdrojovou adresou. Novší API zdroj ponúka jednoduchšie

definíciu dát vo formáte JSON a vylepšené zabezpečenie najmä v prípade, kedy by sa bezpečnostným tokenom mohla dostať ďalšia osoba. Jednotlivé parametre sú uvedené v tabuľke číslo 7. Nové tokeny majú kratšiu životnosť a využíva sa tu autentifikácia pomocou OAuth 2. Obnovovacie tokeny sa prenášajú menej často a tak sa znižuje riziko, že budú odchytené. Jednotlivé notifikácie je jednoduchšie zasielať aj pre ďalšie platformy ako napríklad iOS alebo Android vďaka formátu. Na produkčnom prostredí môže prebiehať komunikácie len pomocou šifrovaného spojenia, využitím HTTPS. (Schéma 6)




Obr. 6: Flow chart pre prijímanie notifikácií

```
request({url: 'https://fcm.googleapis.com/fcm/send', method: 'POST', json: true
    , headers: {'Authorization': 'key=' + SERVER_AUTH_KEY,'Content-Type': '
    application/json'
},
body: {'notification': notification, 'registration\_ids': regIds}
}, function (error, response, body) {
// Obsluha odpovede
}});
```

Výpis 8: Odoslanie požiadavku na Google FCM

# 5 Spracovanie asynchrónnych udalostí vo forme notifikácií

Oproti implementácií rozhrania, ktoré využíva externú službu pre doručovanie push notifikácií sa zameriam na implementáciu rozhrania pre konkrétnu prípadovú webovú aplikácie. Komunikácia bude prebiehať v rámci aplikácie, doručovanie bude v prípade, že je aktuálny používateľ online. Udalosť, správa nepredstavuje štandardný push protokol – notifikáciu ako bolo tomu v predošlej kapitole. Jednotlivé poznatky môžu byť použité v rámci implementácie real-time komunikácie, ktorá dodatočne prenáša len potrebné množstvo dát.

Komunikácia bude prebiehať pomocou rozhrania pre zasielanie a prijímanie správ. Jednotlivé správy budú ukladané v rámci fronty. Prípadové použitie predstavuje aktívnych používateľov webovej aplikácie, ktoré prehliadajú konkrétny obsah a na stránke aplikačného serveru dochádza k vyvolaniu udalosti, ktorá má za cieľ informovať používateľov s preddefinovanými dátami.

V rámci zamerania sa práce na body ako sú škálovanie a ukladanie dát sa zameriam na použitie fronty pomocou technológie RabbitMQ, ktorá bude obsahovať požiadavky a následne časť, ktorá bude jednotlivé správy spracovávať a komunikovať medzi používateľom a serverom. Vypracovanie bude používať implementáciu skriptu v NodeJS a dodatočné ukladanie v nerelačnej databáze MongoDB. Komponenta broker pre komunikáciu medzi koncovým klientom a frontou predstavuje komunikačný endpoint, ktorý bude znázorňovať princípy systému pre prenos správ s dodatočnou aplikačnou logikou. Pri výbere jednotlivých technológií bol kladený dôraz na dostupnosť existujúcich nástrojov ako knižníc pre komunikáciu s databázou, spojenie na prenos a odoberanie správ spolu s ich perzistenciou a jednoduché zobrazenie popisovaného riešenia v programovacom jazyku. Zvolenie RabbitMQ prináša do riešenia prvky dostupnosti viacerých protokolov, možnosti nastavenia smerovania a ukladania správ spolu, škálovanie a s dodatočným nástrojom pre monitorovanie a znázornenie výsledkov. Spĺňa tak požiadavky pre prenos menších správ podľa neskorších funkčných požiadaviek, kedy spracovanie nad dátami prebieha v aplikácií alebo koncovom bode. Výhodou použitia MongoDB sú podobne možnosti škálovania s nerelačným formátom dát, ktorý môžeme jednoducho priebežne meniť a dobrým pomerom výkonu z hľadiska čítania a zapisovania a taktiež prípadných agregačných dotazov nad dátami. Použitý programovací jazyk NodeJs bol vybraný z hľadiska rýchlosti implementácie a jednoduchej čitateľnosti výsledného riešenia a podporou behového prostredia na platforme Linux.

## 5.1 Funkčné požiadavky

V rámci definovania funkčných požiadaviek budem hovoriť o strane klienta a backendovej časti systému. Samotný klient predstavuje script umiestnený na webovej stránke, na ktorej sa aktuálne nachádza daný používateľ. Používateľ navštevuje produktovú stránku v danej kategórie a v prípade vyvolania akcie na backendovej časti systému príma webovú notifikáciu. Skript na strane klienta vytvára spojenie so serverom pri načítaní konkrétnej stránky. Medzi funkčné

požiadavky webového portálu, ktorý sa zaoberá predajom a inzerciou automobilov zaradíme prípady zasielanie notifikácie pri nasledovných udalostiach.

- Udalosť vytvorenia nového inzerátu v konkrétnej kategórií.
- Udalosť prijatia privátnej správy používateľom, reakcia na inzerát.
- Špeciálne ponuky ako napríklad zníženie ceny automobile či dodatočná výbava v cene.

Vytvorené rozhranie v podobe koncového bodu bude zobrazovať nadhľad na samotným fungovaním systémov pre prenos správ a architektúrou. Funkčným požiadavkám v rámci aplikačnej logiky bude možnosť vytvorenia požiadavku na odoslanie webovej notifikácie skupine používateľov či konkrétnemu používateľovi. Potvrdenie doručenia správy nie je podmienkou. Funkciou úložiska je ponúknuť perzistenciu pre prijaté správy a ich zaradenie do konkrétnej kategórie, z ktorej môže neskôr čerpať dáta koncový bod pre používateľov. (Schéma 7)

Koncový používateľ, ktorý využíva webový prehliadač pre prehliadanie webových stránok a spĺňa technické požiadavky pre podporu webových notifikácií dostáva informačné notifikácie generované systémom, ktoré sa zobrazujú v pravom dolnom rohu obrazovky. Notifikácia zobrazuje náhľadový obrázok s nadpisom a krátkym popisom. Po akcií kliknutia je používateľ presmerovaný na konkrétnu webovú lokalitu. K prijímaniu správ dochádza počas aktivity na webových stránkach.

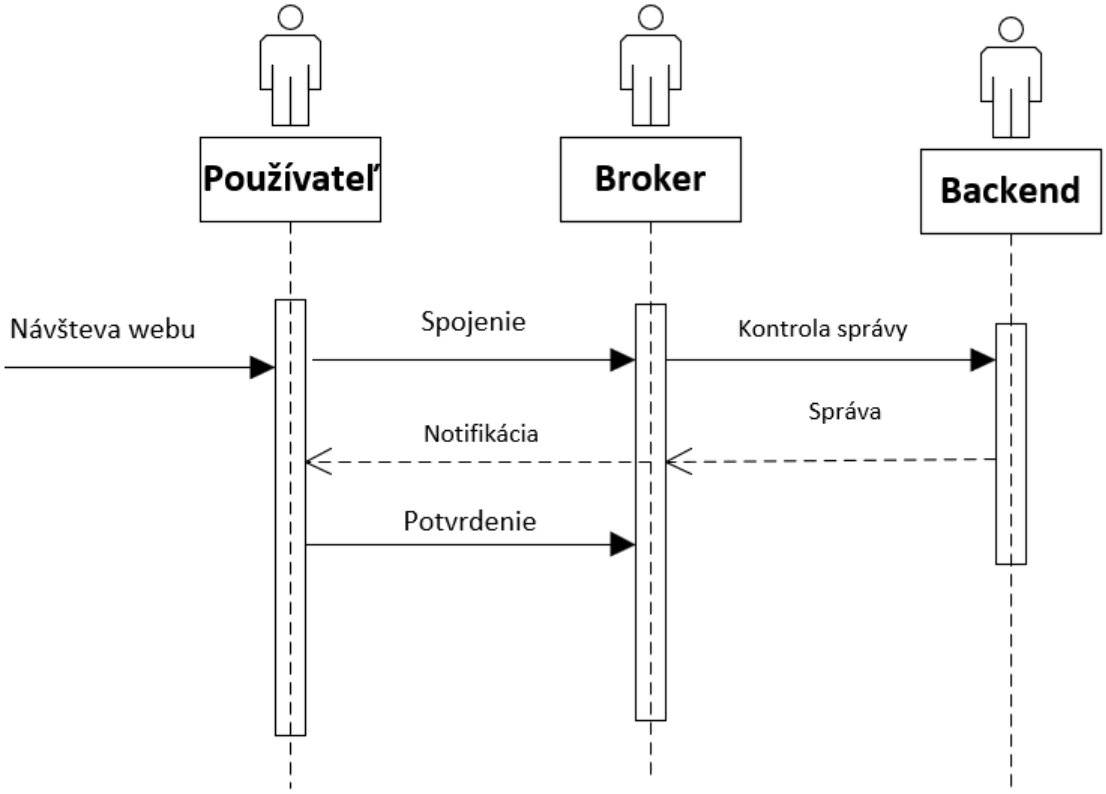
## 5.2 Technická špecifikácia

V rámci technickej špecifikácie znázorním technické požiadavky pre funkčnosť navrhnutého riešenia. Pre funkčnosť klienta je nutné zabezpečiť podporu v samotnom prehliadači pre zobrazovanie notifikácií nakoľko riešenie počíta aj s rozšírením do budúcnosti pre existujúce PUSH služby, notifikácie, ktoré by fungovali aj mimo samotnú webovú aplikáciu. (Tabuľka 8)

Tabuľka 8: Podpora webových notifikácií v prehliadačoch

| IE | Edge | Firefox | Chrome | Safari | iOS Safari | Opera Mini | Chrome Android |
|---|---|---|---|---|---|---|---|
| | | | 49 | | | | |
| | | | 70 | | | | |
| | | | 71 | | **11.4** | | |
| **11** | 18 | 65 | 72 | 12 | **12.1** | **V** | 71 |
| | | 66 | 73 | 12.1 | 12.2 | | |
| | | 67 | 74 | | | | |
| | | | 75 | | | | |

*V – všetky verzie, tmavo vyznačené verzie nepodporujú web notifikácie* [24]



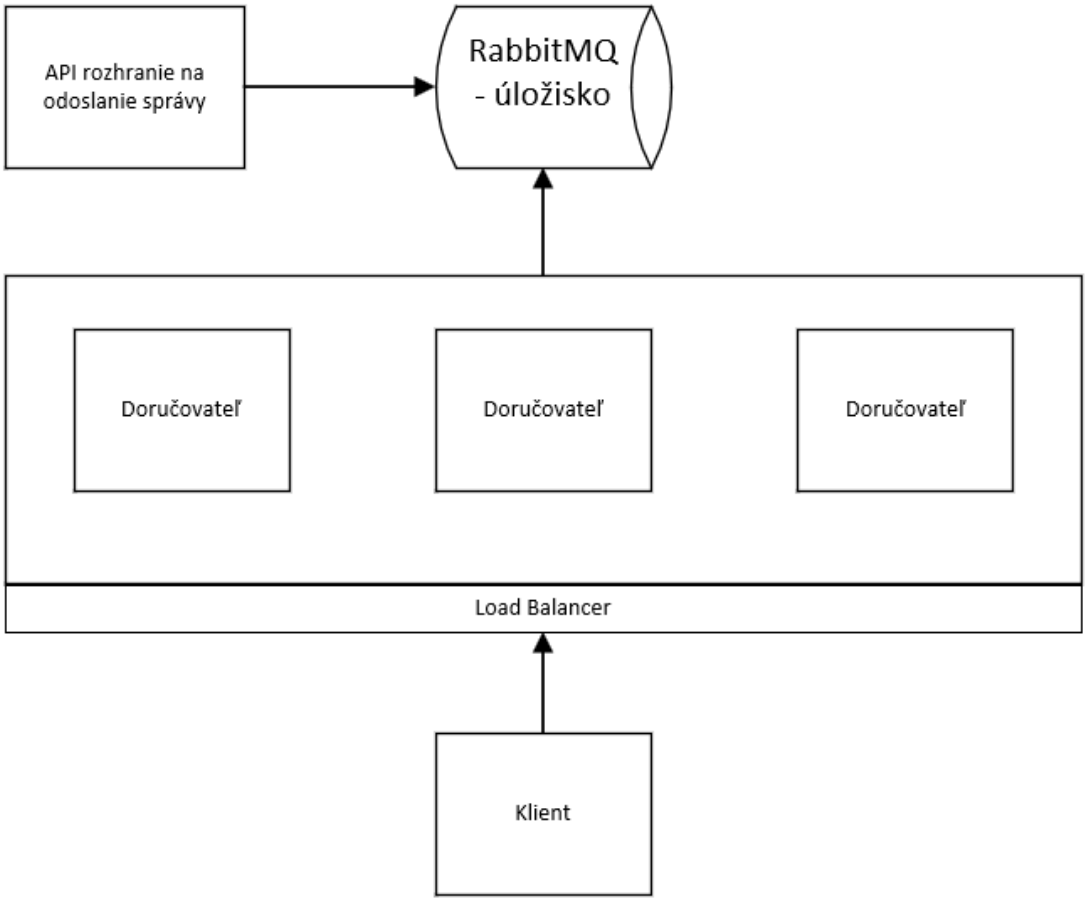

Obr. 7: Sekvenčný diagram prenosu notifikácie

Časť systému nazvaná ako broker, predstavuje nezávislú jednotku bez priamych závislosti na iné časti systému. Broker komunikuje spolu s frontou a obsluhuje taktiež existujúce spojenia od používateľov, ktorý sú naň pripojený. Po technickej stránke bude zabezpečené linuxové prostredie s podporou behu NodeJS skriptov. Samotnú časť je možné následne spúšťa ako distribuovanú službu, ktorej popis bude zohľadnený v samostatnej kapitole o možnostiach rozširovania z hľadiska škálovania. Spustenie RabbitMQ servera pre perzistenciu bude podobne využívať linuxové prostredie. Z hľadiska konfiguráciu a fungovania bude prijímať a uchovávať správy, ktoré budú ukladané do kategorizovaných front. Následne z týchto front budú správy odoberané jednotlivými doručovateľmi "brokermi" pre koncových používateľov.

Pre vytvorený systém zložený z jednotlivých komponent budem zvažovať pokročilé možnosti škálovania. Okrem koncových klientov, ktoré predstavujú v uvedenom príklade unikátne zariadenia môžeme vytvoriť jednotlivé služby, ktoré budú zahrňovať niekoľko bežiacich inštancií, ktoré budú splňovať potrebnú redundanciu a v prípade vyššieho počtu požiadaviek aj zvýšený výkon pri rozkladaní záťaže medzi jednotlivé body.

## 5.3 Návrh architektúry

Nasledujúci uvedený návrh graficky znázorňuje existujúce komponenty systému spolu s naznačenou komunikáciou, ktorá bude prebiehať. V rámci kapitoly popíšem jednotlivé komponenty z hľadiska fungovania pri spracovaní dát, možnosti pripojenia pre ďalšie služby a popis nefunkčných prvkov, ktoré podajú lepší pohľad na to, ako jednotlivé prvky pracujú. Výsledkom návrhu bude popis, ktorý bude predstavovať dokumentáciu vhodnú pre fázu neskoršej implementácie. Predpokladom pre fungovanie systému je zapojenie podsystémov tak aby dokázali spolu komunikovať a spĺňali jednotlivé funkčné ale aj nefunkčné požiadavky. (Schéma 8)




Obr. 8: Architektúra komponentov pre prenos správ

## 5.4 API rozhranie pre zasielanie správ

Vytvorené rozhranie bude v hlavnej úlohe poskytovať metódy pre zaslanie správy na ďalšiu komponentu, ktorou bude úložisko samotných správ. Vrstva perzistencie dát predstavuje rozhranie, ktoré vytvára kontrakt pre triedy, ktoré ho budú implementovať, obsahovať vopred definované

popisy metód. Aplikačná logika komponenty využíva túto vrstvu pre komunikáciu. Podmienkou implementácie bude možnosť kedykoľvek v budúcnosti zameniť úložisko správ za iné pri čom by nemala byť priamo ovplyvnená aplikačná logika.

Komponenta využíva na komunikáciu HTTP protokol a v rámci neho ponúka REST API rozhranie využitím formátu dát JSON pre telo požiadavku a na svojom vstupe poskytuje koncové body, ktoré využíva časť aplikačného webového backendu, v konkrétom prípade, webový autobazár a predstavuje konkrétneho klienta. Implementácia by mala zaručiť možnosť nových klientov a nezávislosť na prenášaných dátach. Klient môže kedykoľvek zavolať službu, ktorý mu poskytne relevantnú odpoveď na požiadavku.

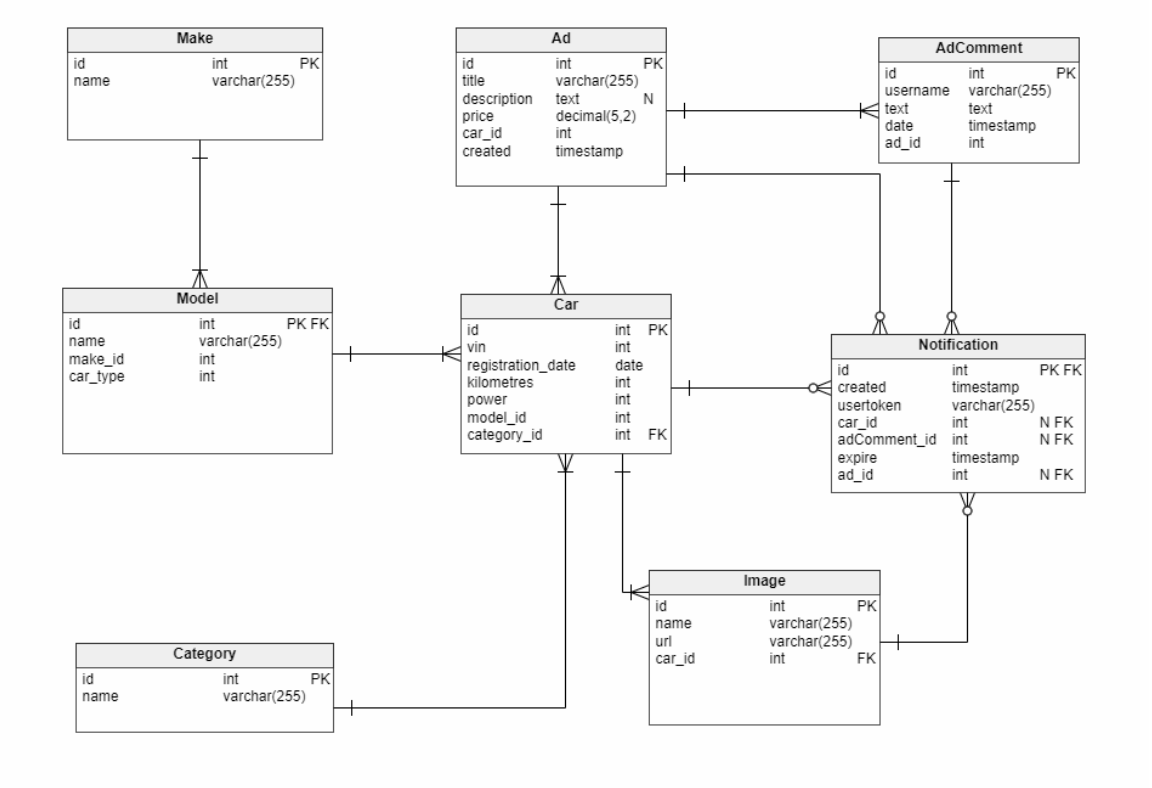
### 5.4.1 Dátový model

Uvedený dátový model znázorňuje zjednodušený návrh databázové modelu, ktoré modeluje entity v rámci inherentného portálu zameraného na predaj automobilov. Zoznam entít obsahuje značku automobilu, modelový tip, model, kategóriu, inzerát, komentáre k inzerátu, obrázky a entitu pre ukladanie zaslaných notifikácií, ktoré môže referencovať viacero ďalší entít. Ukladanie záznamov o zaslaných notifikáciách môže slúžiť pre účely generovanie štatistík, historizácie dát prípadne ďalšieho spracovania. Dodatočná kontrola o tom či bola správa zaslaná môže zabrániť opätovnému zasielaniu rovnakej správy priamo v časti backendu aplikácie. (Schéma 9)

Navrhnutý model by ďalej mohol byť rozšírený o entity parametrov jednotlivých automobilov, používateľov aplikácie, správu oprávnení, záznamy o importoch dát z tretích strán, ktoré predstavujú dodávateľov a predajcov automobilov, záznamy o reakciách na inzeráty, správy medzi používateľmi, tvorbu vlastných zoznamov inzerátov a ďalšie entity pre ukladanie dát z hľadiska analýzy. Dodatočné nové dáta môžu byť podobne ako existujúce z návrhu byť prepojené pomocou vzoru adaptéru do rozhrania na zasielania správ.

## 5.5 Prenos správ a perzistencia využitím RabbitMQ

Centrálna komponentá doručovateľa správ, ktorá v rámci celkovej systémovej architektúry dostáva správy z backendového API rozhrania a následne ponúka možnosť čítať správy odberateľovy na strane koncového bodu. Ponuka potrebné vlastnosti, ktoré zabezpečia prenos správy spolu s jej smerovaním a uložením, možnosti škálovania, monitorovania a konfigurácie vlastností jednotlivých prvkov. Z hľadiska architektúry systému sa zameriam na popis pre navrhnuté webové riešenie. Podľa funkčných požiadaviek musí komponenta uchovávať kategorizované správy, umožniť viacero súčasných spojení a zabezpečiť perzistenciu. Jednotlivé prijaté správy je možné priradzovať do konkrétnych front. Ich názov môže byť definovaný ako UTF-8 formátovaný reťazec dĺžky maximálne 255B, ktorý nemôže obsahovať vyhradený systémový prefix äqt". Unikátne meno je vygenerované v prípade požiadavku s prázdnym argumentom mena fronty. Výsledné názvy front budú zodpovedať účelom ich vytvorenia. Pre príklad, fronta, ktorá obsahuje nové pridané inzeráty má unikátny názov “ads” a podobne pre používateľské správy či kategorizované




Obr. 9: Zjednodušený model pre automobilový inzertný portál

inzeráty. Z hľadiska perzistencie je požiadavkou uchovanie správ aj pri reštarte RabbitMQ. Perzistentná fronta predstavuje konfiguráciu ako "Durable"pričom ďalší režim je "Lazy". Jednotlivé dáta sú uchovávané v pamäti RAM pre vyšší výkon. V prípade nedostatočných systémových zdrojov dochádza k zápisu na disk. Následné čítanie tak môže dosahovať oveľa horšie výsledky. Režim fronty Lazy zapisuje na disk už pri vytvorení či uložení správ, kedy je potrebné uvoľniť pamäť RAM prípadne neexistujú odberatelia správ. Poradie správ vychádza z poradia, podľa ktorého boli prijaté. Prvá prijatá správa ja taktiež ako prvá odoslaná pri čítaní odberateľom. Výnimku môžu tvoriť prioritné fronty. Odosielateľ, v našom prípade API rozhranie môže špecifikovať prioritu správy prípadne samotnú prioritu môže mať aj odberateľ, ktorý má pri čítaní správ následne prednosť. Priorita predstavuje kladné číslo od 1-255 pričom z hľadiská výkonu budem využívať prípadne maximálny level priority 5. Využitie dočasných front je možné využiť pri komunikácií jednotlivých používateľov medzi sebou. Samotné dáta sú štandardne ukladané v databáze a prenášané správy majú notifikačný charakter. Životnosť dočasne fronty končí v prípade, že neexistuje žiaden odberateľ správ. Z pohľadu fungovania systému vieme na základe používateľského prístupu na webové stránky vytvoriť perzistentné spojenie, ktoré bude prijímať správy. Výhodou použitia tohto typu front je jednoduchšie smerovanie v rámci aplikácie a hlavne prenos správy, kedy nečakáme na odobratie iných správ, ktoré aj tak nie sú v konečnom dôsledku určené pre konkrétneho používateľa. Nevýhodou môžu byť vyššie nároky na systémové zdroje.

Z hľadiska výkonu je potrebné zabezpečiť dostatočný výkon podľa konfigurácie pre prenos správ. Klienti vo forme aplikačného rozhrania a koncového bodu, ktorý môžu byť neskôr škálované by mali udržiavať permanentné TCP spojenie a nevytvárať nové pri každom požiadavku. Požiadavkou je taktiež uzatvárať spojenia aby nedochádzalo k uníkom spojenia, kedy je uzavreté na strane klient ale obslužný proces na strane RabbitMQ je stále vytvorený. Pri prenášaných notifikáciách nie je striktnou požiadavkou ich doručenie z čoho vyplýva, že systém nemusí čakať na potvrdenie prijatia odberateľom. Toto nastavenie zvyšuje výkon prenosu správ. Jednotlivé správy tak môžu mať nastavené aj dĺžku uchovania vo fronte, pričom pokiaľ nebudú odobrané v určitom časovom okne, budú automaticky z fronty odstránené. Pri jednotlivých frontách musíme taktiež sledovať ich veľkosť, ktorá môže lineárne rásť v prípade, že strana odberateľa nestíhať spracovávať a naopak na strane aplikačného rozhrania neustále dochádza ku zasielaniu nových správ. Z hľadiska navrhnutej architektúry uvažujeme o fronte s obsahom maximálne 10 000 správ počas štandardnej prevádzky. V prípade využitia viacerých front dochádza k zvýšeniu výkonu nakoľko jedna fronta využíva jedno procesorové vlákno v danom momente.

Priradenie do jednotlivých front pre odberateľov môže byť riešené viacerým spôsobmi. Správa, ktorú odosiela aplikačné rozhranie môže mať špecifikované dodatočné parametre, ktoré ovplyvnia jej priradenie. V prípade definovania priameho názvu fronty dochádza pri smerovaní k priradeniu podľa mena. Dodatočnou konfiguráciou parametrov môžeme dosiahnuť priradenie viacerým frontám. Jedným z parametrov môže byť “exchange”, ktorý zodpovedá bodu za smerovanie. Jednotlivé fronty môžu mať definovaný "binding_key", ktorý zodpovedá priradeniu ku konkrétnej ëxchange". Aplikačné rozhranie môže definovať "routing_key", kedy dochádza na základe uvedenej hodnoty k nasmerovania do príslušných front, ktoré spĺňajú pravidlo. Hodnota môže byť taktiež spracovaná ako predmet, pomocou regulárneho výrazu. Aktuálny smerovač môže byť taktiež konfigurovaný v režime “fanout”, kedy automaticky preposiela správy na všetky príslušné fronty. Dodatočnou konfiguráciou toho ako bude správa smerovaná je definovanie hodnoty v hlavičke správy. Zmenou nastavení tak môže systém pracovať v režime Publish/Subscribe, kedy koncový bod nie je pripojený na konkrétnu frontu ale odoberá správy na základe definovaného predmetu. V prípade viacerých koncových bodov môžeme taktiež určiť množstvo správ, ktoré budú v jednom požiadavku odobrané z fronty. Odoberanie súvisí s nastavenou prioritou a taktiež vlastnosťami koncového bodu, ktoré následne musí túto dávku spracovať. Takúto dávku správ je možné aj potvrdzovať súčasne pokiaľ by bolo povolené potvrdzovanie prebratia správ.

## 5.6 Doručovanie pomocou koncového bodu

Koncový bod, ktorý je umiestnený medzi jednotlivými klientami, ktorý predstavujú používateľov webovej aplikácie a systémom pre prenos správ zohráva z hľadiska architektúry dôležitú úlohu a vytvára v rámci kontextu nezávislý prvok. Pre porovnanie typického modelu Požiadavok-Odpoveď by sme uvažovali o koncových klientoch, ktorý sa v jednom prípade pripájajú na aplikačný backend a snažia sa získať správy. V tomto prípade veľká časť aplikácie pôsobí ako monolit, kedy musí spravovať pripojenia a celý prenos správy alebo využívať na rovnakej vrstve

systém pre prenos správ. V ďalšom prípade klienti obsahujú u seba aplikačnú logiku, pomocou ktorej sa pripájajú priamo na systém prenosu správ a odoberajú jednotlivé dáta. V oboch prípadoch vzniká úzka závislosť na jednotlivých prvkoch. Prípadne škálovanie je veľmi zložité rovnako ako hľadanie chýb či nasadzovanie novej verzie aktuálneho programu.

Navrhnuté rozhranie pre koncový bod je podobné ako aplikačné, ktoré využíva aplikačný backend a pomocou neho zasiela správy. Narozdiel od neho, uvedený koncový bod komunikuje so systémom pre prenos správ, z ktorého odoberá dáta, konkrétne z existujúcich front, dokáže využívať aj aplikačnú logiku, na základe ktorej dokáže vytvárať funkcionalitu smerovača a transformácie dát pre klientov a neposlednom rade spravuje fyzické pripojenia od klientov. Z hľadiska komunikácie so systémom pre prenos správ môže komponenta pracovať vo viacerých režimoch. Tieto boli uvedené v kapitole fungovanie systémov pre prenos dát. V navrhovanom prípade využívame Event-driven architektúru, ktorá vytvára akciu pri uložení správy v konkrétnom prípade systémom RabbitMQ. Koncový bod predstavuje odberateľa správ. Oproti akumulovanému odberateľovi sa líši v tom, že odoberá správy aj v prípade, že nie je pripojený žiaden koncový klient. V prípade, že existuje viacero inštancií koncového bodu, každá príma rovnakú správu nakoľko uvažujeme o odbere správ kedy jednotlivé inštancie odberajú správy na základe predmetu, formou PUB/SUB.

Úlohou komponenty je pracovať aj aplikačnou logikou, ktorá využíva v našom prípade statickú konfiguráciu. Rozhranie na základe dát zo správy, ktoré v riešenom prípade môžu predstavovať unikátny identifikátor inzerátu získava z databázy aktuálny záznam. Tento záznam prechádza časťou, ktorá sa stará o transformáciu dát, kedy vytvárame zjednodušený model definovaný v rámci komponenty pomocou mapovania dát zo zdrojových na cieľové. Na jednotlivé dáta môžu byť aplikované filtre, ktoré postupné menia dáta, ktoré budú na výstupe. V ďalšom kroku je možné na správu aplikovať smerovanie, ktoré zaručí, že správa bude doručená danej cieľovej skupine. Príkladom skupiny môžu byť používatelia, ktorý sa aktuálne nachádzajú v určitej kategórií automobilov alebo naopak nachádzajú na domovskej stránke. Nevýhodou dodatočných krokov s pridanou aplikačnou logikou je nižšia priepustnosť koncového bodu, kedy je potrebný určitý čas pre spracovanie dát a ich následne zaslanie klientovi. Správa klientskych pripojený predstavuje dočasné úložisko jednotlivých spojení, ktoré nepracuje s prípadnými procesmi, ktoré by mohli prebiehať na pozadí klienta aj po jeho odpojení z aplikácia. V prípade zlyhania koncového bodu dochádza k strate spojenia. Komunikácia je obojsmerná, kedy klient nadväzuje spojenie a po jeho úspešnom pripojení ho koncový bod informuje v prípade nových dát. Klientska časť predstavuje vzor observera, kedy naviac môžu samotný klienti odoberať viacero predmetov.

## 5.7 Klientská aplikácia

Podmienkou pre fungovanie klientskej aplikácie je webový prehliadač s možnosťou behu skriptu a aktívneho povolenia pre zobrazovanie webových notifikácií, ktoré získava od koncového používateľa. Zo zadania skript pracuje v rámci aplikácie a nepredstavuje proces na pozadí v podobe

SW. Úlohou klientskeho skriptu je nadviazanie spojenia s koncovým bodom pričom môže byť doplnený o jednoduchú aplikačnú logiku. Tá môže určovať predmety, ktoré bude klient odoberať a na ich základe príjmať konkrétne správy. Prijatá správa vo vopred dohodnutom formáte je následne spracovaná a je vyvolaná webová notifikácia. Klient nevytvára dodatočné požiadavky na koncový bod kedy by synchrónne čakal na jeho odpoveď.

## 5.8 Zebezpečenie

Pri zabezpečení hovoríme o šifrovanom spojení, zabezpečení dátovej integrity či samotnej autorizácií a autentifikácií. Výsledok riešenia sa zameriava na architektonické prvky pri prenose správ. V prípade zabezpečenia hovoríme o dodatočných možnostiach rozšírenia. U uvedenom návrhu je možnosť zabezpečenia spojenia pre API rozhranie, RabbitMQ a koncový bod. V prípade API rozhrania je možnosť zabezpečenia na úrovni prístupu, ktoré môže typicky využívať existujúce riešenia ako autentifikáciu spolu s TLS nad ktorou pracuje SSL. Tá môže byť realizovaná napríklad JWT tokenmi. Ďalšie možnosti sú jednoduchšia HTTP autentifikácie či metódy OAuth 1.0, OAuth 2.

RabbitMQ ponúka používateľskú autentifikáciu spolu s autorizáciou voči existujúcim prvkom ako sú fronty, smerovače a ďalšie prvky. Okrem používateľského mena a hesla pridáva možnosť využitia X.509 certifikátov prípadne overovanie voči autentifikačnému backendu. Štandardným zabezpečením je obmedzenie pripojenia pre používateľa „quest“, ktorý sa môže pripojiť len v rámci lokálnej siete. V rámci autorizácie systém zavádza zoznam zdrojov a možnosť priradiť konkrétnemu používateľovi prístup k operáciám ako sú konfiguráciu, zápis a čítanie. Autorizovať používateľa je možné aj voči samotným predmetom, ktoré by chcel odoberať. Dostupné možnosti pre zabezpečenie sú formou HTTP, LDAP a AMQP protokolov.

Medzi klientom a koncovým bodom je možnosť kontroly existujúcej relácie, kedy môžeme filtrovať napríklad registrovaných používateľov a návštevníkov. Z hľadiska zabezpečenia sa snažíme o to aby spojenie bolo nadväzované len zo zdrojovej aplikácie overením hlavičiek požiadavku. Dôležitou podmienkou pre zabezpečenie je šifrovanie spojenia kedy môžeme využiť protokol HTTPS.

## 5.9 Škálovanie, testovanie a integrita

Z uvedeného návrhu sa snažíme vytvoriť nezávisle softvérové jednotky, ktoré nebudú mať medzi sebou silné previazanie. Výsledkom toho tak môžeme jednotlivé inštancie nezávisle škálovať a môžu bežať v rozličných lokalitách a s rozličnou hardvérovou výbavou. Návrh počíta aj s možnosťami jednoduché nasadenia, ktorý vytvorí hotovú službu. Možnosťou pre zabezpečenie požiadaviek je využitie existujúcej kontajnerovej technológie, ktorá vytvorí uzavreté prostredie pre beh a pridá dodatočnú úroveň zabezpečenia. Vo výslednom prípade tak môžu koncové body obsahovať rozličné klientske pripojenia, systém pre prenos správ môže byť jednoducho spustený na viacerých inštanciách, taktiež nazvaných ako shardoch, kedy dochádza k replikácií dát a

taktiež aj samotné API rozhranie, kedy uvažujeme o tom, že ho využíva viacero rôznych aplikácií ako bránu pre zaslanie správy.

Aplikačné testovanie môže byť na najnižšej úrovni, ktorá tu predstavuje samotné komponenty. To môže byť realizované pomocou unit testov, kedy testujeme funkcionalitu jednotlivých metód či integrácií, kedy test zahrňuje aj externé prvky ako napríklad databázy. Zo zadania uvažujeme o splnení podmienok integrácie, kedy považujeme test za úspešný v prípade, že je správa zaslaná aplikačným backendom doručenia klientom. Samotný formát dát, ďalšie kroky ako sú smerovanie, transformácie a iné by museli byť realizované pomocou aplikačných testov, ktoré predstavujú jednotlivé testovacie sady spolu s testovacími metódami.

# 6 Implementácia systému pre asynchrónny prenos správ

Nasledujúca kapitola sa bude zaoberať popisom implementácie jednotlivých komponentov, ktoré boli popísané v návrhu architektúry navrhovaného systému. Funkčnosť implementácie je orientovaná na webový portál zaoberajúci sa inzerovaním automobilov spolu s používateľmi, ktorý vykonávajú akcie ako sú vytváranie inzerátov, reakcie a samotné prehľadanie stránok. Výsledkom implementácie bude systém pozostávajúci z viacerých komponent, ktorý zdieľa doménovú logiku a v prípade inej aplikácie je možné ho prispôsobiť na tieto účely. V popise budú zahrnuté jednotlivé vrstvy, postup implementácie a použité nástroje a vzory.

## 6.1 Použíte nástroje a vzory

Dôležitým cieľom implementácie bolo vytvoriť rozhrania, ktoré zaručia kontrakt pre jednotlivé triedy, ktoré ich budú implementovať. Pri výbere programovacie jazyka NodeJs boli zohľadnené existujúce knižnice, ktoré je možné použiť pri implementácií z hľadiska vytvorenia prehľadného a čitateľného riešenia, ktoré dá vo výsledku konkrétnu informáciu o tom ako funguje popisovaný systém pre zasielanie správ a ako by ho bolo možné v budúcnosti rozširovať.

### 6.1.1 NodeJs

Uvedený nástroj môžeme nazvať knižnicou – frameworkom pričom najčastejšie je možné sa stretnúť s pojmom behového prostredia. NodeJS je ako prostredie postavené nad V8 Engine pričom využíva programovací jazyk Javascript. Vďaka prostrediu dokáže byť takýto javascriptový kód spustený aj mimo klasický prehliadač, v ktorom bol najčastejšie používaný práve pred príchodom NodeJS. Samotný kód predstavuje prvky skriptovacie programovacie jazyka, ktorý je prekladaný do strojového jazyka a následnej je interpretovaný systémom. Medzi hlavné výhody NodeJS patri model spracovania, ktorý neblokuje vstupno-výstupné operácie a predstavuje architektúru založenú na vyvolávaní systémových udalostí. Za opak môžem uviesť model, kde sa o jednu IO operáciu stará konkrétnej jedno vlákno procesoru. Pri neblokujúcom vstupno-výstupnom modely taktiež odpadá možnosť vzniku takzvaných mŕtvych zámkov nad systémovými zdrojmi. O spracovanie sa stará slučka udalostí, ktorá sa stará o vykonanie vstupu a jednotlivé spätná volania sú ukladané do fronty. Vďaka asynchrónnemu spracovaniu sa hodí v spracovávanom prípade pre riadenie HTTP požiadaviek či iných spojení s požiadaviek na konkurentné spracovanie.

### 6.1.2 Typescript

Jazyk, ktorý predstavuje nadmnožinu pôvodného Javascriptového jazyka a pred samotným spustením počas vytvárania aplikácie je pôvodný kód prekladaný do Javascriptu. Typescript je vydávaný ako open-source pod záštitou Microsoftu, podporuje štandard ECMAScript 3 a vyššie a pridáva najmä výhody pri používaní pre vývojárov aplikácie. Samotný Javascript predstavuje dynamicky typovaný jazyk, kedy typ premennej je určený až na základe hodnoty počas behu.

Okrem statického typovania je možné využiť aj typické prvky, ktoré môžeme nájsť v iných populárnych programovacích jazykoch ako sú OOP triedy, rozhrania a genericita. Okrem týchto prvkov je bohatá podpora jazyka v rámci vývojárskych prostredí, ktoré umožňujú kód krokovať a upozorňovať na chyby vzniknuté počas kompilácie hlavne čo sa týka typov premenných či kontroly null hodnôt. Pred spustením aplikácie sa spúšťa zadefinovaný skript, ktoré preloží typescriptové súbory so sufixom .ts na .js súbory. Nakoľko je v tomto prípade Javascript podmnožinou Typescriptu, môžeme ľubovoľný kód priamo spustiť ako .ts súborový typ.

### 6.1.3 NPM

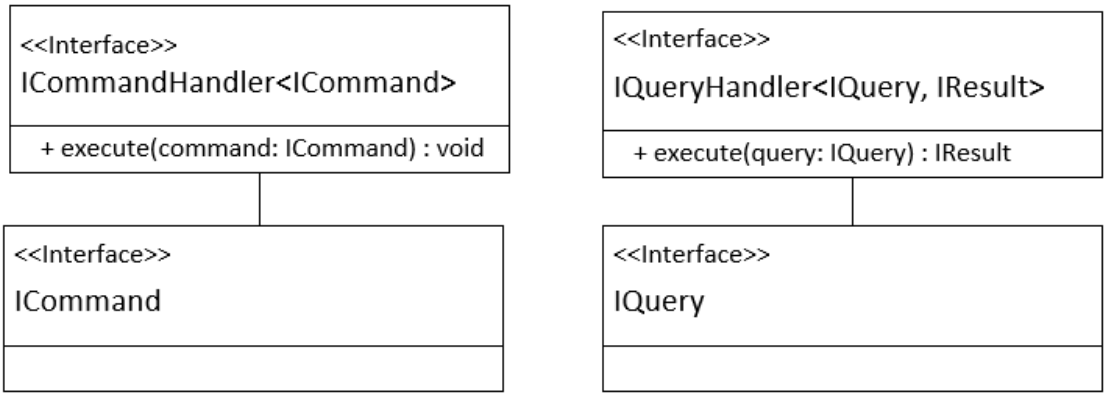
Správa balíčkov pre NodeJs, ktorý ponúka ekosystém pre ukladanie dát, ktoré obsahujú zdrojové kódy a jednotlivé moduly spolu so správou závislosti medzi jednotlivými balíčkami. Projekt, ktorý využíva správcu npm musí obsahovať package.json súbor, ktorý obsahuje informácie o samotnom projekte ako je meno, verzia a ďalšie dodatočné informácie spolu so závislosťami, ktoré predstavujú názvy ďalších balíčkov – projektov spolu s uvedenou verziou, ktoré je využívaná v rámci zdrojového projektu. Samotná NPM platforma ďalej obsahuje webové rozhranie pre prehliadanie balíčkov a spolu s nimi pre popis a štatistické informácie pričom sa kladie dôraz na odhaľovanie prípadných záškodných balíčkov, ktoré by mohli obsahovať malware či iných nechcený kód. Ďalším prvkom je CLI nástroj, ktorý ponúka príkazy pre konzolu vďaka, ktorým môžeme konfigurovať projekt či inštalovať – mazať jednotlivé závislosti.

### 6.1.4 CQRS a Event sourcing

Návrhový vzor, ktoré v svojej podstate oddeľuje model pre perzistenciu na dve časti. Jednou z nich je časť na čítanie a druhá sa stará o zapisovanie. Výsledkom bude jednoznačne oddelenie ako tak závislosti tak aj zodpovednosti jednotlivých prístupov. Vzor predstavuje opak k menej komplexnejšiemu prístupu, ktorý predstavuje CRUD model, kedy je jeden záznam spracovaný modelom pričom medzi jednotlivými krokmi môže dochádzať k rozličným operáciám pričom pokiaľ by boli operácie oddelené, mohli by z hľadiská návrhu pracovať s dáta – vlastnosťami entity, ktoré naozaj potrebujú. Pri použití vzoru je možné pracovať s fyzický oddeleným úložiskom na model pre zapisovanie a druhý pre čítanie. Uvedeným prístupom je možné napríklad predchádzať konfliktom v rámci konkurentných spojení, ktoré využívajú rozličné operácie. Prípadná autorizácia môže byť taktiež rozličná pre zápis a čítanie.

Medzi ďalšie výhody okrem oddelenia zodpovednosti modelov patria možnosti návrhu rozhraní pre jednotlivé domény systému v prípade DDD techník, kedy každá doména obsahuje svoje oddelené modely, možnosť škálovať oddelene zápis a čítanie najmä v prípadoch kedy počet dotazov na získavanie dát vo veľkom počet presahuje tie na vkladanie a v neposlednom rade využitie Event sourcingu z hľadiska integrácie medzi rozličnými systémami. Nevýhodou je väčšia komplexnosť riešenia, kedy v prípade jednoduchšej biznis logiky či správy dát je vhodnejšie zvoliť jednoduchší model.

Spomenutý Event Sourcing v rámci CQRS predstavuje funkčné riešenie ako pracovať s aktuálne vykonávanými akciami v prípade modelu čítania. Predstavuje vyvolanie udalosti, najčastejšie na konci príkazu, ktorý ma za úlohu zaznamenať stav aktuálnej entity. Stav môže byť uložený v nezávislom úložisku, v ktorom sa nachádza aj história predošlých stavov. Takýmto prístupom môžeme získať možnosti zachytenia aktuálneho stavu, atomické spracovanie, flexibilnú prácu s dátami či odstránenie závislosti medzi systémami, ktoré odoberajú správu na základe typu vytvorenéj udalosti.(Schéma 10)



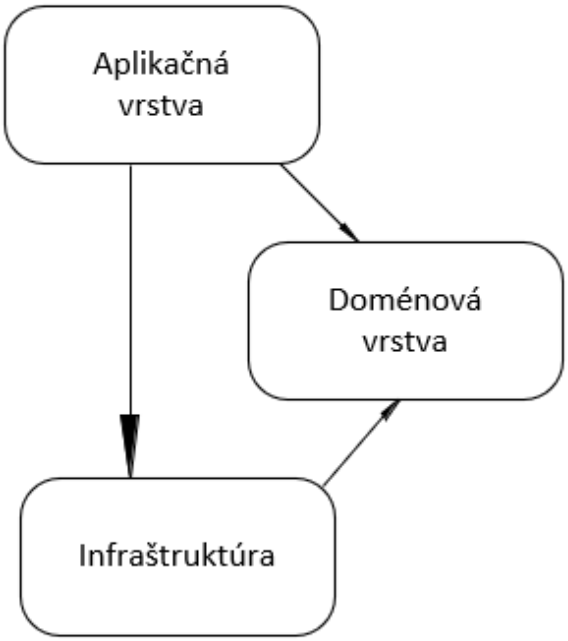

Obr. 10: Ukážka rozhraní pre command a query modely

### 6.1.5 Vrstvy aplikácie

V rámci DDD rozdeľujeme aplikáciu na niekoľko častí. Dôležitým prvkom pri návrhu je oddelenie jednotlivých kontextov aplikácie, chápaných ako logických prvkov, dosiahnuť nižšie previazanie, kedy dosiahneme ohraničený model, ktorý môže zahŕňať okrem aplikačného kódu aj prácu s databázou. (Schéma 11)

- **Aplikačná vrstva** obsahuje logiku samotnej aplikácie, je závislá na doménovej vrstve avšak žiadna iná vrstva nie je na nej závislá. Obsahom sú rozhrania – kontrakty pre ostatné vrstvy.
- **Doménová vrstva** obsahuje logiku pre danú doménu, môžu to byť entity, typy definované ako enum či rôzne aplikačné výnimky, kontrakty pre repozitáre a podobne.
- **Infraštrúktúra** vrstva, ktorá zabezpečuje prístup k externým zdrojom (napríklad služby), implementuje rozhrania z aplikačnéj vrstvy a tak sprístupňuje konkrétnu implementáciu.

Vytvorený projekt, ktorý obsahuje všetky časti riešenia obsahuje nasledujúcu infraštruktúru, ktorá je prispôsobená pre vybraný programovací jazyk. Príkladom môžu byť niektoré entity, ktoré boli implementované ako rozhrania pre možnosť zaručiť kontrakt aj v prípade priraďovania anonymného objektu. Jednotlivé celky obsahujú vrstvy podľa doménového diagramu.

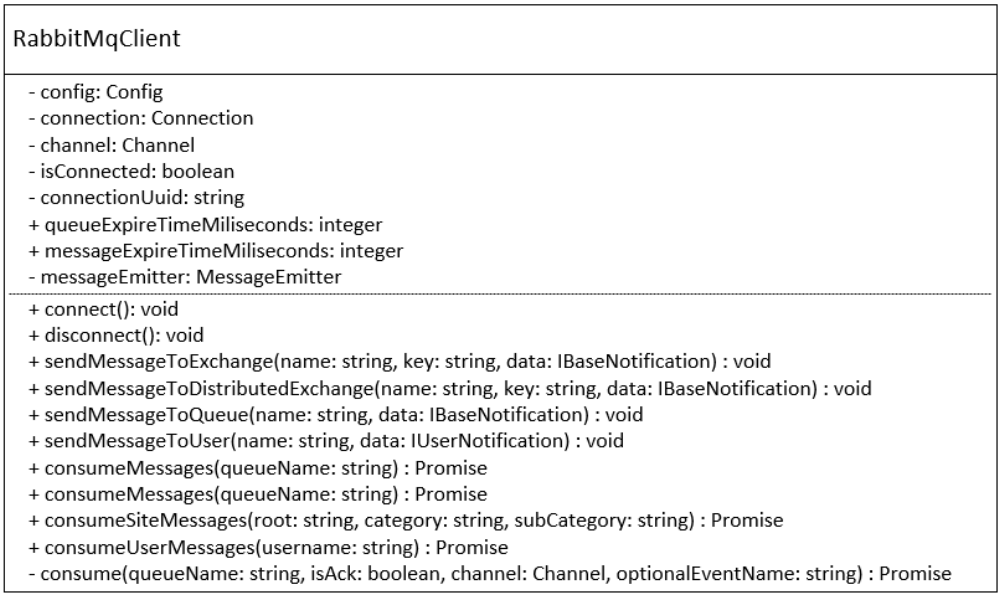



Obr. 11: Diagram DDD služby

- **Koreň projektu** – obsahuje vstupný skript index.ts, ktorý pre zjednodušenie zaobaluje logiku aplikácie. Využíva parametre zo vstupu príkazového riadku a podľa toho rozhoduje či bude spustená API aplikácie alebo endpoint. Ďalej tu sú obsiahnuté konfiguračné súbory pre Typescript a NPM.
- **Klient** – obsahuje ukážkový HTML súbor spolu so skriptom pre pripojenie k endpointu s dodatočnou externou knižnicou.
- **Core** – koreňový projekt, ktorý zahŕňa funkcionalitu zdieľanú medzi ďalšími projektami ako sú napríklad klienti pre RabbitMQ či MongoDB. Štruktúra odpovedá doménovému dizajnu.
- **API, Endpoint** – v koreni projektu obsahujú súbor, ktorý zodpovedá za beh konkrétnej aplikácie. Aplikačná vrstva obsahuje podľa CQRS príkazy a dotazy spolu s vlastnými udalosťami, doménová vrstva jednotlivé entity a infraštruktúra implementáciu na základe kontraktov z aplikačnej vrstvy.

### 6.1.6 RabbitMQ klient

Implementovany klient predstavuje formu repozitáru, ktorý využíva NPM knižnicu amqplib pre potrebu komunikácie s RabbitMQ. V rámci pripojenia je vytvorené unikátne spojenie, nad ktorým sú ďalej vytvárané komunikačné kanále. Tie zabezpečujú napríklad preberanie či odoslanie správ. Implementácia využíva dodatočne aj konfiguráciu pre fronty a ich napojenie na smerovače správ (exchanges). (Schéma 12)



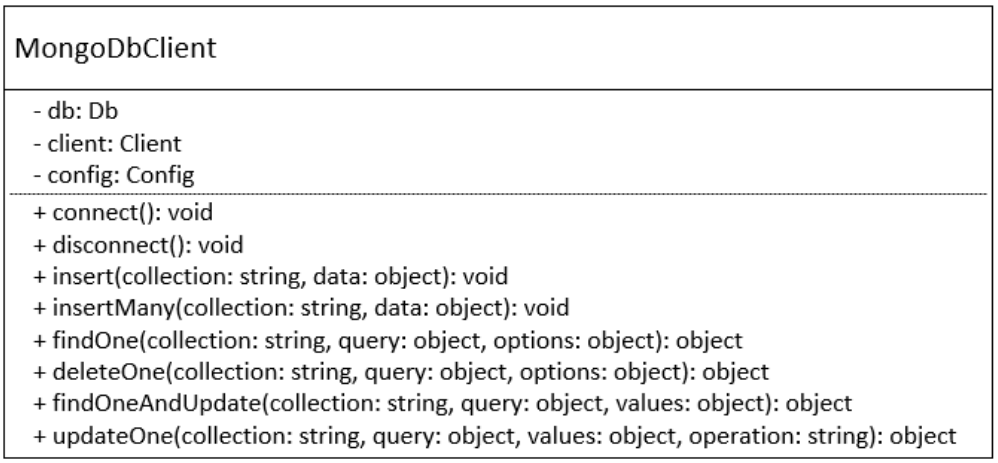

Obr. 12: UML triedny diagram RabbitMQ klienta

### 6.1.7 MongoDB klient

Podobne ako predošlý klient obsahuje funkcionalitu pre komunikáciu, v tomto prípade s NoSQL databázou. Využíva NPM knižnicu MongoDB. Pre potreby vkladanie väčšieho počtu záznamov je volaná metóda insertMany, ktoré dostáva potvrdenie o uložení až na konci transakcie. V rámci aktualizácie záznamov je možné upravovať v modely konkrétneho záznamu aj vlastnosti objektov. Aktualizácia tak môže vykonávať operáciu set alebo push, ktorá buď vloží vlastnosť do objektu alebo ho úplne nahradí. Metóda findOneAndUpdate zaručuje atomickú aktualizáciu v prípade, že by k rovnakému dokumentu pristupoval ďalší klient. (Schéma 13)

## 6.2 API komponenta

Implementácia ponúka REST API rozhrania, pomocou ktorého je možné odosielať správy bez toho aby komponenta alebo odosielateľ vedeli o tom ako bude správa ďalej spracovaná a nakoniec doručená. Rozhranie využíva populárny webový NodeJs framework Express, ktorý ponúka možnosť jednoduché vytvorenia serveru s dostatočným výkonom pre spracovanie požiadaviek.



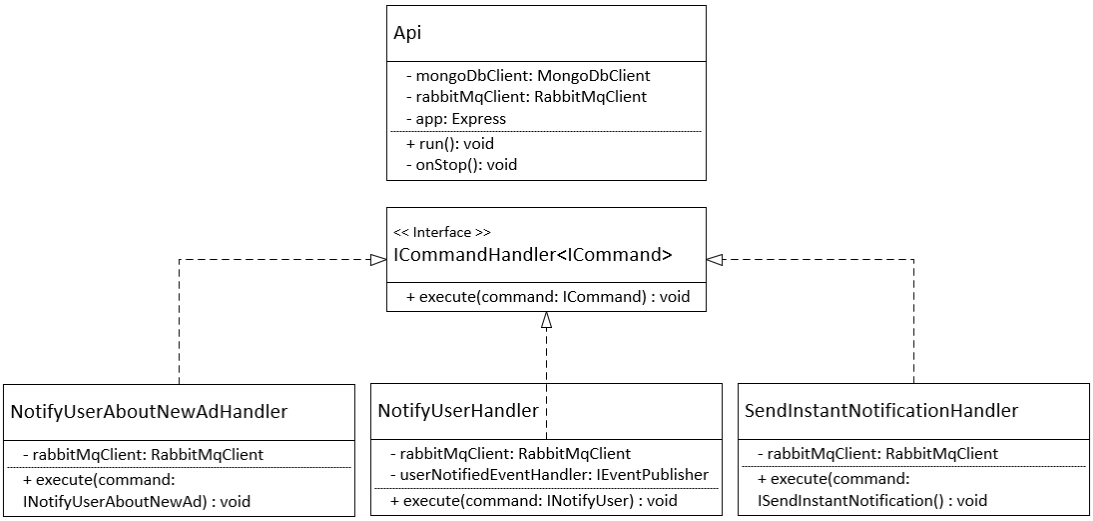

Obr. 13: UML triedny diagram MongoDB klienta

Implementácia obsahuje jednoduchšiu logiku, kedy registruje potrebný middleware (prostredníkov) pre konkrétne koncové API body spolu s obslužnou funkciou, ktorá je vyvolaná formou spätného volania. Komunikácia využíva v testovacom prostredí využíva HTTP protokol zatiaľ čo v produkčnom je odporúčaná zabezpečená verzia so šifrovaním spojenia. (Schéma 14)

Tabuľka 9: Prehľad koncových bodov API rozhrania

| Názov koncového bodu | Metóda | Telo správy | Popis |
|---|---|---|---|
| /notification/ad/ | POST | id: integer | Odošle notifikáciu u o novom inzeráte |
| /notification/user/ | POST | {senderName: string, recipientName: string, message: string} | Odošle notifikáciu prijímateľovi o novej správe. |
| /notification/ | POST | {title: string, message: string, image: string, icon: string, url: string, rootSite: string, category: string, siteSubCategory: string} | Odošle vopred definovanú notifikáciu, prijímatelia sú definovaný na základe cesty (root, category, subCategory) |

Pri odosielaní notifikácií na uvedené koncové body z tabuľky 9 nás nezaujíma či bude vôbec doručená, prípadne koľko klientov správu obdrží. Podobná informácia je využívaná skôr na štatistické účely sledovania kliknutí. Používateľské správy by však mohli byť v budúcnosti sledované z hľadiska toho či boli vôbec a kedy odoslané. Odoslanie používateľskej notifikácie vyvoláva uda-




Obr. 14: UML triedny diagram funkcionality API rozhrania

losť, ktoré predstavuje “Event Source”. Log notifikácií je ukladaný v rámci MongoDB databáze. Formát správy obsahuje:

- Názov udalosti: ÜSER_NOTIFIED"
- Ľubovoľné dáta: payload
- UserId: používateľské ID
- UUID: unikátne ID notifikácie pre možnosť identifikácie v rámci systému

Za doručenie a vyvolanie udalosti je následnej zodpovedná komponenta koncového bodu. Nakoľko na existujúce riešenie nie je pripojených viac služieb, nie je publikovaná priama správa, ktorá by automaticky takéto služby upozornila.

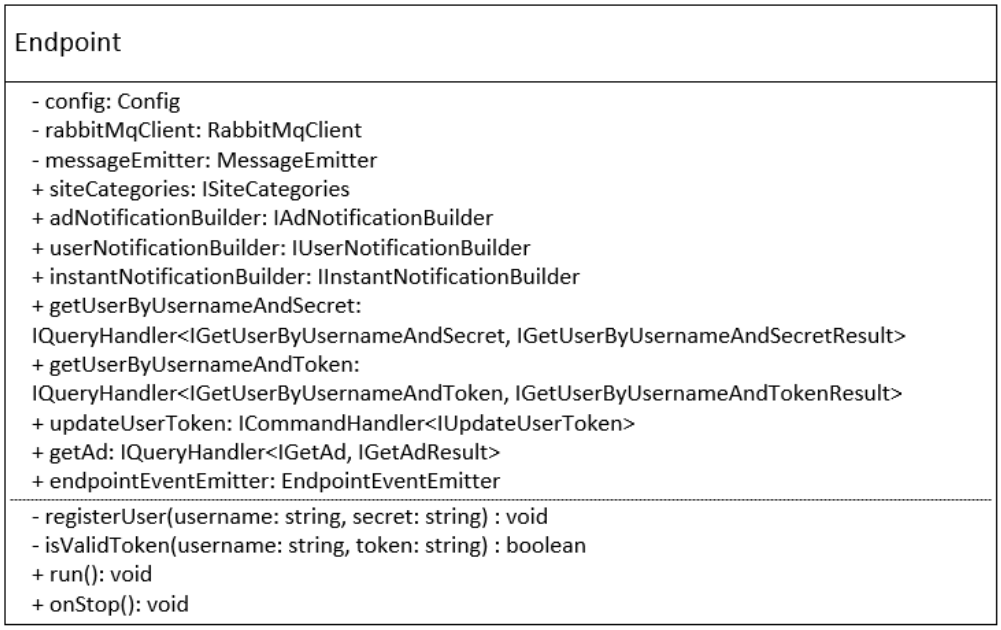
```
this.app.post('/notification/user/', (req: any, res: any) => {
try {
let handler: ICommandHandler<INotifyUser> = new NotifyUserHandler(this.
    rabbitMqClient, this.mongoDb);
let notification: INotifyUser = new NotifyUser(req.body.senderName, req.body.
    recipientName, req.body.message);
handler.execute(notification);
res.sendStatus(200);
}
catch (e) {
```

```
res.sendStatus(500);
}
});
```

Výpis 9: Ukážka implementácie koncového API body spolu s volaním funkcie

## 6.3 Endpoint komponenta

Implementovaná komponenta predstavuje hlavný prvok v rámci architektúry systému, ktorý napĺňa požadované funkčné požiadavky pre doručenie správy ku klientom s dodatočnou aplikačnou logikou. Na jednej strane komunikuje s frontou správ a na druhej funguje ako server pre klientov. Uvedený UML diagram je znázornený v schéme číslo 15 spolu so schémami číslo 21 a 24.




Obr. 15: UML triedny diagram endpoint komponenty

### 6.3.1 Komunikácia s klientami

Endpoint komponenta funguje ako server, ktorý komunikuje s klientom pomocou WebSocket protokolu. Ten predstavuje obojstrannú komunikáciu na vytvorenom TCP spojení a je podobný HTTP protokolu. Je výhodou je nízka réžia pri prenose, kedy dodatočné metadáta tvoria len niekoľko bajtov. Pred vytvorením spojenia dochádza k nadviazaniu, dohode o komunikácií medzi oboma stranami. Okrem verzie ws je možné komunikovať aj pomocou wss typu protokolu, ktorý je zabezpečený a využíva TLS. Použitie je vhodné na komunikáciu v reálnom čase.

Pre implementáciu bola zvolená knižnica socket.io, ktorá predstavuje nadstavbu tradičných websocketov. Do odosielaných packetov pridáva dodatočné dáta pre zaručenie komunikácie a automaticky spravuje komunikačné protokoly napríklad v prípade, že nie je možné vytvoriť websocketové spojenie (napríklad proxy server medzi stanicami), môže využiť iné, napríklad AJAX long polling či klasické HTTP. Pre komunikáciu je nutné využívať knižnicu na strane servera aj klienta. Výhodou je jednoduchá práca pri vyvolávaní udalosti, kedy môžeme komunikovať s jednotlivými klientami alebo odosielať správy ako broadcast a reagovať na pripojenie či stratu spojenia s klientom.

Na strane servera boli vytvorené nasledovné udalosti:

- **onConnection** – udalosť vyvolaná pri vytvorení nového klientskeho spojenia. Dochádza k overeniu spojenia, kedy sa kontroluje či nadviazané spojenie nie je duplicitne. Duplicitné spojenie v rámci webovej aplikácie môže nastať napríklad v prípade, že používateľ stránku otvorí vo viacerých oknách. Klientske spojenie je uložené do pamäte v podobe objektu. Po nadviazaní spojenia dochádza k vyvolaniu udalosti registrácie na klientovi.
- **onRegister** – udalosť vyvolaná klientom, ktorý sa chce registrovať. Overenie používateľa je na základe používateľského mena a tajného hesla – reťazcu, ktorý je v databáze uložený. Po registrácií je klient zaregistrovaný pri prijímanie inherentných notifikácií.
- **onListenUserMessage** – udalosť vyvolaná klientom, kedy chce odoberať privátne správy.
- **onListenCategory** – udalosť vyvolaná klientom, kedy chce odoberať správy pre konkrétnu kategóriu. Kategória je špecifikovaná kľúčom, ktorý je vytvorený na základe parametrov (koreňová stránka, kategória, podkategória).
- **onDisconnect** – udalosť vyvolaná pri odpojení klienta. Dochádza k uvoľneniu systémových zdrojov a používateľ je odstránený z poslucháčov udalostí, ktoré odoberal.

Jednotlivé BPNM diagramy, ktoré modelujú priebeh udalosti sú zobrazené v prílohe diplomovej práce. Overovanie spojenia je využívané najmä pri privátnych správach, kedy sa kvôli bezpečnosti generuje opätovne používateľský token, ktorý je predstavovaný ako náhodný reťazec. Platnosť sa overuje pred odoslaním správy používateľovi, kedy v prípade, že je neplatný dochádza k vyvolaniu klientskej udalosti “renew” a používateľ sa musí znova registrovať zaslaním údajov, kedy mu je vystavený nový platný token. Jednotlivé akcie ako volania na databázu či frontu sú spracovávané asynchrónne. Jednotlivé funkcie sú označené kľúčovým slovom “async” kedy telo funkcie následne môže obsahovať kľúčové slovo “await” pre asynchrónne volania. Z hľadiska fungovania tento vzor v prípade, že narazí na volanie funkcie, na ktoré čaká, predáva riadenie program ďalej a do samotnej funkcie sa vráti až po získaní výsledku. Na prácu s udalosťami je využívaný NodeJs EventEmitter, ktorý obsahuje metódy pre správu poslucháčov a definovanie udalostí. Model rozhraní a tried, ktoré sa využívajú pre odoslanie koncovej push notifikácie je zobrazený v prílohe diplomovej práce.

O vytvorenie koncovej push notifikácie sa stará konkrétna "builder"trieda, ktoré v rámci svojho vzoru využíva jednu metódu build, ktorá príma parameter na základe implementovaného generického rozhrania a môže vykonávať ľubovoľnú aplikačnú logiku. Implementovanú triedu je možné vďaka rozhraniu kedykoľvek zameniť.

Pred odoslaním push notifikácie klientovi je aplikovaný filter, ktorý je opäť vďaka rozhraniu možné zameniť. Filter môže rozhodnúť o tom či bude správa odoslaná prípadne pozmeniť dáta. Spolu s vytvorenou notifikáciou, ktorá je výstupom konkrétneho buildera príma na vstup parameter používateľa, ktorému bude doručená. (Schéma 16)

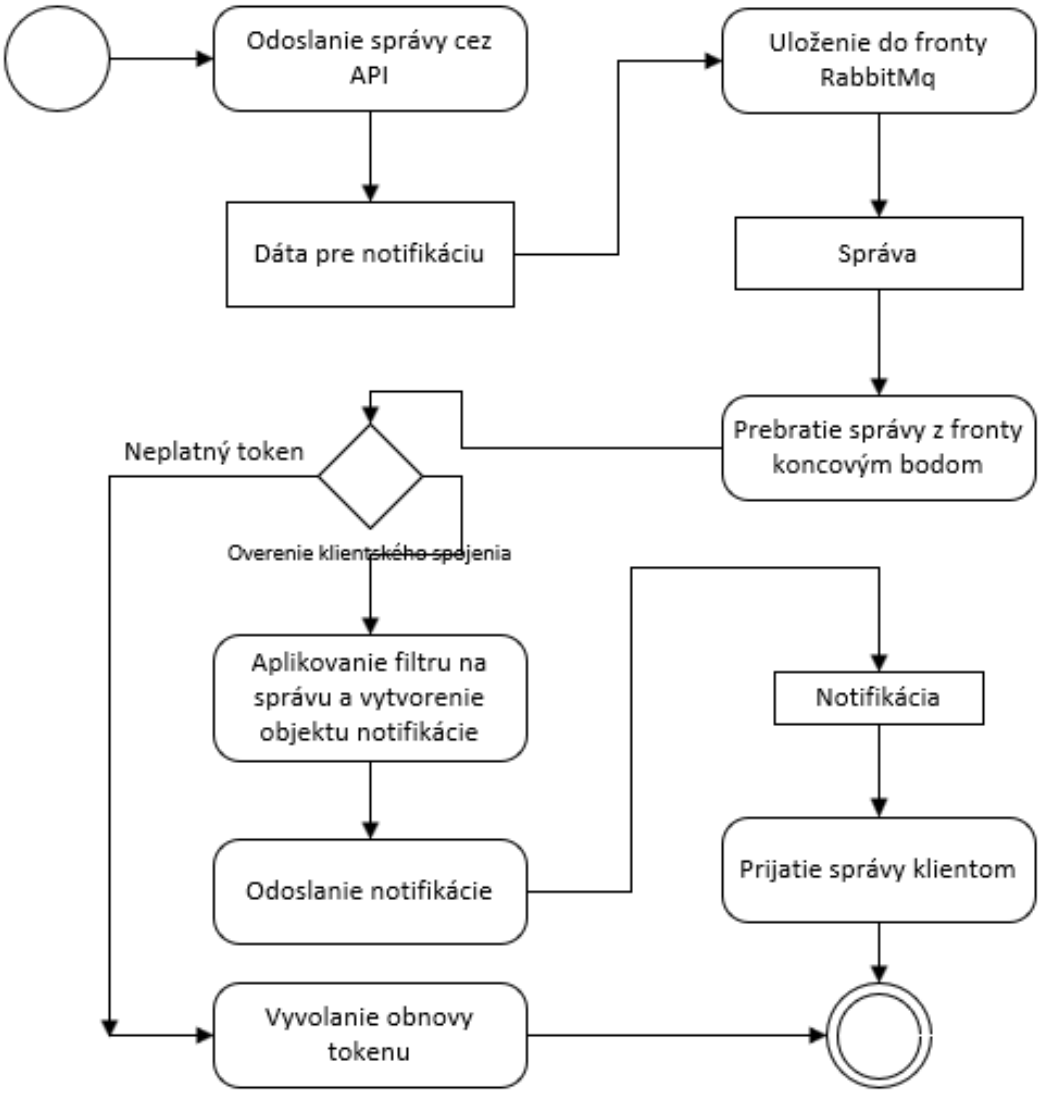
```
// Funkcia pre vytvorenie konkrétnej notifikácie a jej odoslanie
user.categoryListener = function(notification: IInstantNotification) {
const user = connections.get(client);
let pushNotification = endpoint.instantNotificationBuilder.build(notification);
endpoint.endpointEventEmitter.emit('notification', client, user,
    pushNotification);
};

// Registrácia poslucháča udalostí
endpoint.rabbitMqClient.messageEmitter.addListener(categoryEventName, user.
    categoryListener);

// Vyvolanie udalosti
endpoint.rabbitMqClient.messageEmitter.emit('someCategory', notificationObject)
    ;

// Registrácia poslucháča udalostí pre odoselanie správ klientom
this.on('notification', function(client: any, user: IEndpointUser, notification
    : IPushNotification) {
setImmediate((user, notification, client) => {
notification = emitter.pushNotificationFilter.apply(user, notification);
if(notification != null) {
client.emit('notification', notification);
}
});
});
```

Výpis 10: Ukážka registrácie a volaní udalosti s asynchrónnym spracovaním




Obr. 16: UML aktivitný diagram zobrazujúci priebeh doručovania správy

## 6.4 Konfigurácia RabbitMQ

Systém RabbitMQ má v rámci architektúry úlohu doručovateľa správ medzi API a koncovým bodom. Jeho druhou úlohou je vytvárať perzistentnú vrstvu pre správy. Z hľadiska implementácie bola nakonfigurovaná jedna inštancia spolu s rozhraním manažmentu systému, ktorý ponúka prehľad o aktuálnom stave systému pričom zahrňuje údaje o vytvorených frontách, zmenárňach, počet uložených, potvrdených a odoberaných správ spolu s grafmi. Systémové informácie zobrazujú údaje o počte procesov, veľkosti pamäte, ktorú využíva systém a diskovom priestore. V základnej konfigurácií taktiež vidíme aktívne spojenia a jednotlivé vlastnosti je možné konfigurovať aj cez toto webové rozhranie. (Schéma 17)

Podľa návrhu bolo potrebné vytvoriť konfiguráciu pre prenos 3 typov správ. Pre jednotlivé zmenárne je možné konfigurovať unikátne meno, typ (topic, headers, fanout, direct), ktorý upravuje správanie fronty, vlastnosť fronty či má udržiavať správy – zapisovať na disk aby boli dostupné aj v prípade reštartu či chyby systému a možnosť automatického zmazania napríklad v prípade, že neexistuje žiaden odberateľ, v tomto prípade fronta, alebo ďalšia zmenáreň, ktorá je namapovaná na túto konkrétnu. V prípade fronty sú konfigurované parametre ak meno, automatické zmazanie a vlastnosť udržiavania dát spolu s TTL časom, ktorý hovorí o tom kedy jednotlivé správy exspirujú a taktiež celá fronta. Dodatočne je možné špecifikovať veľkosť jednotlivých správ, prioritu či zápis na disk pomocou lazy módu.

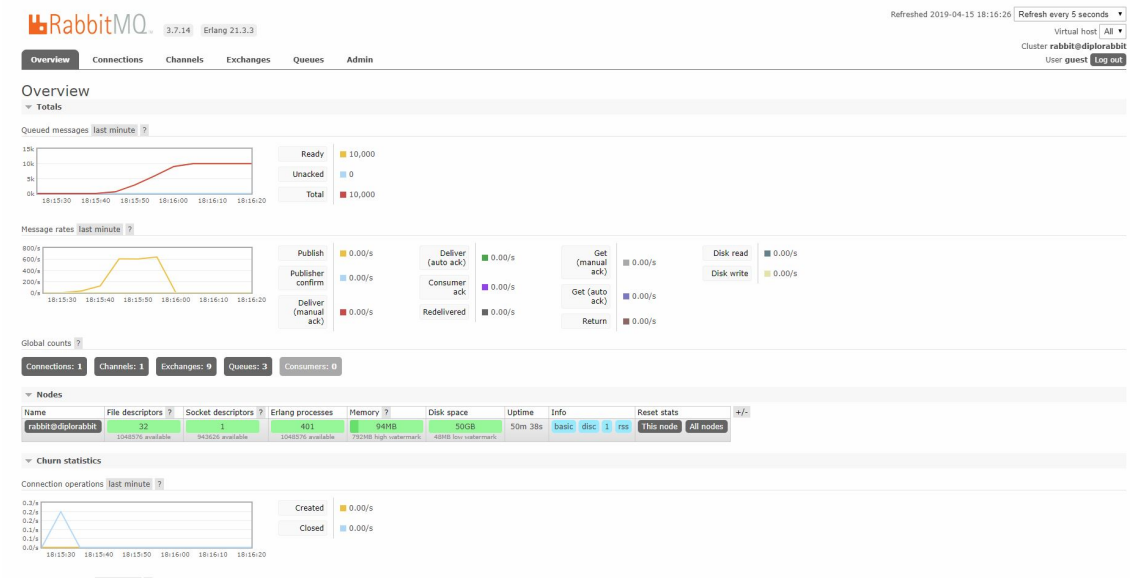
**Štandardné inzerentné notifikácie** je možné zasielať do fronty s názvom äds", kedy existujte len jeden odberateľ (koncový bod) alebo je jedno, ktorá skupina klientov obdrží konkrétnu notifikáciu. Pre distribuované spracovanie je však vytvorená zmenáreň "notification", na ktorú je možné mapovať jednotlivé fronty vo formáte äds"+ unikátny identifikátor koncového bodu. V prípade doručenia takejto správy do zmenárne je spôsobom "fanout"rozoslaná všetkým frontám, ktoré su na ňu na namapované.
**Používateľské notifikácie** predstavujú formu priameho spojenia, avšak nejedná sa o komunikáciu napríklad formou RPC ale vytvorenie unikátnej fronty pre príjemcu správy. Správy spolu s frontou majú nastavenú dočasnú životnosť, štandardne nastavenú na jednu hodinu, kedy je zohľadnené, že webový používateľ nie je vo väčšine prípadov aktívny celý čas.
**Instantné notifikácie** sa odlišujú od klasických inzerentných formátom dát, ktorý je definovaný už pri odoslaní backendom cez rozhranie API a tak nepodliehajú prípadnej aplikačnej logike na strane koncového bodu. Cez dané rozhranie je možné odosielať ľubovoľné správy s definovanými parametrami. Jednotlivé notifikácie majú naviac definované parametre ako koreňová stránka, kategória a podkategória. Tieto atribúty vytvárajú smerovací kľúč. Správy sú prijímané v rámci RabbitMQ zmenárňou s názvom ïnstant", ktorý na základe kľúču smeruje správy do príslušných front, ktoré sú namapované spolu s konkrétnym kľúčom.

Príklad kľúču:

- *.*.* - správa je smerovaná do všetkých front
- cars.*.* - správa je smerovaná do front s kľúčom "cars"
- cars.bmw.m3 – správa je smerovaná do front s kľúčom "cars", "bmw", "m3"

## 6.5 Implementácia klienta

Webový klient bol vytvoreny formou ukážky jednoduchej statickej webovej HTML stránky. Na stránke sa nachádza skript, ktorý komunikuje s koncovým bodom. Z funkčných požiadaviek vychádzame pri riešení z navrhovanej webovej aplikácie, ktorá pracuje s konkrétnym používateľom




Obr. 17: Ukážka administrátorskeho rozhrania RabbitMQ

a môže definovať rozličné kategórie. V prípade inzeretného automobilového portálu to môžu byť kategórie ako osobné autá – nákladné vozidlá. Pre odoberanie súkromných správ je nutné pri nadviazaní spojenia odoslať aj bezpečnostný kľúč, ktorý môže byť predstavovaný heslom používateľa. Pre znázornenie sú uvedené schémy číslo 22 a 23.

Klientský skript definuje následné udalosti:

- **connect** – udalosť vyvolaná pri úspešnom pripojení
- **disconnect** – udalosť vyvolaná pri odpojení serverom
- **renew** – udalosť vyvolaná v prípade, že je klient pracuje s exspirovaným tokenom
- **registered** – udalosť vyvolaná pri úspešnej registrácií
- **listening-userMessage** – udalosť vyvolaná pri úspešnej registrácií poslucháča udalostí na notifikácie podľa aktuálnej kategórie
- **failed-listening-userMessage** – udalosť vyvolaná pri chybe registrácie poslucháča
- **notification** – udalosť vyvolaná pri prijatí notifikácie klientom, obsahuje samotné dáta
- **category** – udalosť vyvolaná pri zmene kategórie

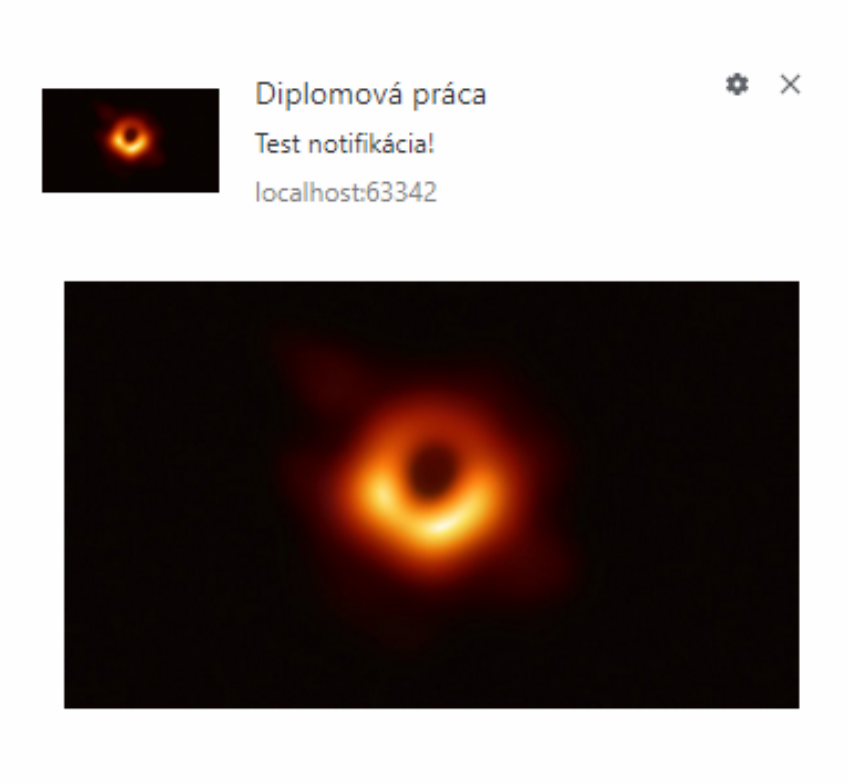
Podmienkou pre prijatie notifikácie je podpora v prehliadači a taktiež podpora websocketov, ktorá využíva knižnicu socket.io podobne ako na serveri. (Schéma 18)

```
var notification = new Notification(data.title, {
icon: data.icon,
image: data.image,
body: data.body,
});

if(data.url != undefined) {
notification.onclick = function (event) {
event.preventDefault();
window.open(data.url, '_blank');
};
}

setTimeout(notification.close.bind(notification), data.duration * 1000);
```

Výpis 11: Ukážka vyvolania notifikácie v prehliadači




Obr. 18: Ukážka zobrazenia notifikácie v prehliadači

## 6.6 Nasadenie a možnosti škálovania

Okrem zabezpečenia prenosu správ vytvorenou implementáciou bola taktiež vytvorená topológia pre prenos správ, ktorá podporuje distribuované spracovanie viacerými inštanciami. Počas implementácie som využíval jednotlivé databáze spustené ako virtuálne kontajneri použitím nástroju Docker.

### 6.6.1 Docker pre prácu s kontajnermi

Ten ponúka jednouchý spôsob ako rýchlo vytvoriť obraz vybranej aplikácie, nasadiť ju do takzvaného kontajneru a spustiť. Kontajner predstavuje virtualizovanú jednotku, ktorá zahŕňa rôzne závislosti a časti aplikácie, ktoré potrebujú medzi sebou komunikovať tak aby vedela fungovať ako jeden celok. Oproti klasickým virtualizovaným strojom sa líši v návrhu architektúry, kedy pracuje priamo nad hostiteľským systémom, kde sa nachádza prostredie Dockeru. Typické virtualizované systémy využívajú vlastný hypervízor, ktorý pridáva vrstvu vlastného operačného systému. Docker naviac prináša vrstvu zabezpečenia a zahŕňa správu siete medzi jednotlivými kontajnermi prípadne službami a hostiteľským operačným systémom.

Z návrhu vychádzame o vytvorení jednotlivých komponent, ktoré dokážu pracovať samostatné a je ich možné škálovať do väčších celkov. Nad skupinou inštancií môže byť následne load balancer, ktorý rozkladá záťaž a výsledkom je tak zvýšenie výkonu ale taktiež aj spoľahlivosti, kedy v prípade výpadku či chyby aplikácie môžu fungovať ďalšie inštancie.

### 6.6.2 Škálovanie RabbitMq

Pri škálovaní konkrétne RabbitMq sa zaoberáme vytvorením redundantných front, ktoré budú predstavovať kópiu hlavnej master fronty. Jednotlivé inštancie tvoria klaster pričom sa odberateľ správ nezávisle pripája na konkrétnu frontu, ktorá je však určená loadbalancerom, ktorý sa nachádza pred samotným klasterom. Ďalším prvkom je potreba potvrdzovania správ, kedy toto potvrdené bude slúžiť pre jednotlivé inštancie ako záruka, že správu môžu vymazať z fronty, ktoré predstavuje kópiu. Problémom môžu byť príliš veľké fronty, kedy synchronizácia zaberie dlhší čas prípadne ak pracujeme s nerezistentnými správami. V prípade, že by došlo k výpadku master inštancie počas synchronizácie, môžeme tak stratiť dáta. Jednou z lepších možnosti je zaručiť potvrdzovanie doručovania a prijímanie pripojení na konkrétnej inštancií až po jej úplnej synchronizácií.

### 6.6.3 Topológia RabbitMq

Pri návrhu topológie, ktorá zahŕňa odosielateľa správ, zmenárne, fronty a odberateľov správ sa nám ponúka niekoľko možnosti ako a komu doručovať jednotlivé správy. Medzi dôležité faktory zaraďujeme vyváženie doručovania medzi jednotlivých odberateľov, poradie spracovania správ, záťaž na smerovanie a spôsob ako spracovávať chybné prípadne nedoručené správy.

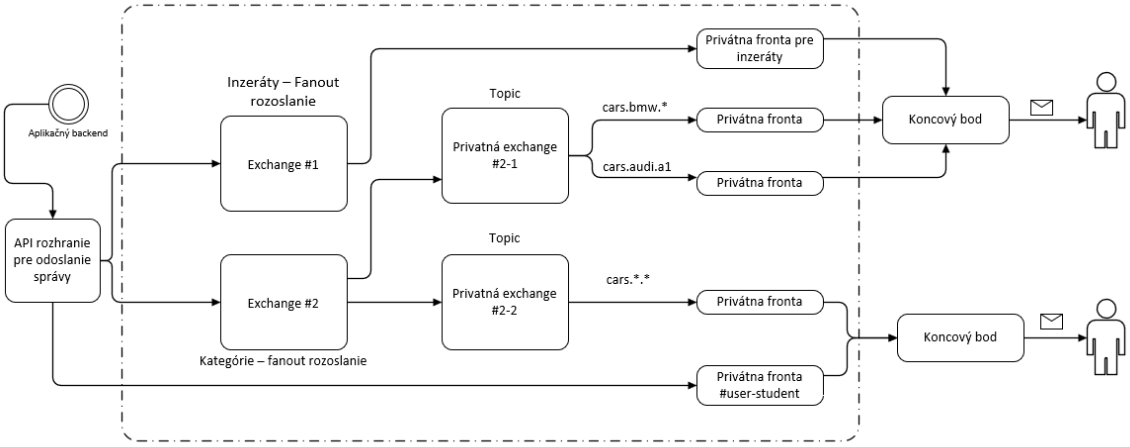
V prípade definovania x odberateľov správ z konkrétnej fronty štandardným spôsobom nemôžeme zaručiť rovnomerné odoberanie. Ak by sme na rôzne stavové typy definovali len jednu vyhradenú frontu, v tomto prípade nezaručíme ani poradie spracovávania. Výsledkom toho je nerovnomerná záťaž spracovávania na odberateľoch a neexistuje scenár v prípade chyby či výpadku správ. Medzi možné riešenia patria využitá rôznych typov zmenárni a front.

- Priame spojenia medzi zmenárňou a frontou, kedy definujeme mapovanie na základe pevného kľúča. Výsledkom je rýchle smerovanie – smerovania avšak nerieši to problém odberateľov u konkrétnej koncovej fronty.
- Spracovanie na základe predmetu správy alebo atribútu v hlavičke. Vieme dynamicky pridávať mapovania a upravovať tak správanie pri smerovaní správy. Nevýhodou môžu byť vyššie systémové náklady pri veľkom počte smerovaní a prípadných zmien mapovania.
- Smerovania správ na základe hash hodnoty. Dokážeme zaručiť poradie vykonávania správ, kedy sú koncové fronty mapované na zmenárňou podľa konkrétnej hash hodnoty, ktorá sa vypočíta podľa správy. Hash hodnota však nemusí zaručiť rovnomernú distribúciu v prípade, že je nekvalitný kľúč.
- Zmena nastavení front, ktoré predstavujú dĺžku životnosti samotných front, správ či prioritu. Na jednotlivých frontách je taktiež možné konfigurovať maximálny limit správ, ktoré môžu byť uložené. V prípade, že životnosť správy vyprší alebo je iným spôsobom odstránená, môže byť uložená do alternatívnej fronty či zmenárne, ktorá sa postará o jej distribúciu.

Pri návrhu topológie sa snažíme znížiť náklady na smerovanie a zaručiť rovnomerné rozloženie medzi jednotlivé fronty a koncových odberateľov. V uvedenom príklade pracujeme so správami, ktoré sú kategorizované, doručované každému odberateľovi a používateľské, ktoré sú doručené výhradne jednému odberateľovi. Na poradí doručenia správ nezáleží. Výsledkom je topológia, ktorá ma niekoľko vrstiev a kategorizované správy spolu s štandardnými sú smerovaná do privátnych zmenárni a front, ktoré sú konfigurované ako dočasné spolu s časovou dĺžkou uchovania správ pre jednotlivé koncové body nakoľko v uvedenej webovej aplikácií majú notifikácie informačný charakter a nepredstavujú stav častí systému. (Schéma 19)

### 6.6.4 Úvod do práce s Dockerom

Nakoľko kontajnerové technológie a problematika vývoju softvéru so zameraním sa na jeho nasadenie, zaistenie stabilného chodu samotnej služby a skrátenie času pri zmenách predstavujú samostatnú kapitolu ako postupovať, tak v nasledujúcom texte popíšem využívané vlastnosti a načrtnem možnosti spustenia výslednej implementácie.




Obr. 19: Ukážká topológie smerovania správ

**6.6.4.1 Tvorba obrazu a spustenie kontajneru** Predstavuje súbor s inštrukciami ako zostaviť výsledný kontajner. Jednotlivé inštrukcie sú zhlukované do vrstiev a môžu predstavovať inštaláciu operačného systému či nástrojov. Oficiálne úložisko takýchto "obrazov" pre vytvorenie kontajneru je Docker Hub, odkiaľ je možné sťahovať či nahrávať vlastné súbory.

Na uvedenom príklade môžeme vidieť stiahnuté obrazu pre RabbitMq a spustenie inštancie kontajneru.

```
docker pull rabbitmq:management
docker run -it -d --hostname diplorabbitmq --name rabbitmq -p 5672:5672 -p
    15672:15672 -v rabbitmq:/var/lib/rabbitmq rabbitmq:management
```

Výpis 12: Vytvorenie docker kontajneru zo stiahnutého obrazu

- run – označuje spustenie kontajneru
- prepínače -it -d predstavujú vytvorenie interaktívneho terminálu a spustenie na pozadí
- hostname definuje názov pre konkrétnu inštanciu RabbitMq pre prípad, že by bola súčasťou clusteru
- –name označuje názov kontajneru, v prípade neuvedenia názvu dôjde k použitiu vygenerovaného názvu
- prepínač -p sprístupňuje port na ľavej strane zo strany hostiteľského zariadenia a mapuje ho na port vnútri kontajneru
- prepínač -v mapuje virtuálnu diskovú jednotku "volume" na cieľový adresár a zabezpečuje uloženie dát aj v prípade zmazania kontajneru

- rabbitmq:management označuje názov obrazu, časť za dvojbodkou predstavuje konkrétnu verziu, v uvedenom príklade management obsahuje nainštalované administračné rozhranie

Pre vytvorenie docker obrazu pre implementované komponenty môžeme vytvoriť vlastný súbor, ktorý bude obsahovať inštrukcie pre vytvorenie kontajneru.

```
FROM node:latest
WORKDIR /app
COPY package.*.json ./
COPY . .
RUN npm install
CMD ["npm", "start"]
```

Výpis 13: Ukážka vlastného docker obrazu

V prvom kroku pomocou FROM vytvárame vrstvu na základe existujúceho obrazu, ktorý vydáva samotný NodeJs a obsahuje operačný systém s behovým prostredím. WORKDIR vytvára pracovný priečinok, nasledovaný COPY príkazmi kopíruje súbory z hostiteľského zariadenie do vytváraného obrazu. Vrstva vytvorená príkazom RUN inštaluje npm baliký tak aby boli zabezpečené závislosti definované v package.json pre projekt. Posledný CMD príkaz definuje konzolový príkaz, ktorý sa prevedie po spustení kontajneru.
Vytvorenie obrazu pre spustenie kontajneru a samotné spustenie je možné pomocou príkazov, ktoré sú podobné ako v predošlom príklade, prepínač -t označuje názov obrazu.

```
docker build -t myApiImage .
docker run --name api -p 3000:3000 myApiImage .
```

Výpis 14: Vytvorenie docker kontajneru z vlastného obrazu

#### 6.6.4.2 Škálovanie služieb

V rámci škálovania a dosiahnutia vyššieho výkonu a stability aj v prípade výpadku niektorej z inštancií je možné vytvárať v jednom prípade z jednotlivých obrazov služby a v ďalšom klaster tvorený z jednotlivých fyzických systémov. Pre vytvorenie služby, ktorá zaobaľuje niekoľko kontajnerov, ktoré medzi sebou komunikujú je možné použiť nástroj docker-compose. Ten využíva YML formát, kde definujeme jednotlivé vlastnosti pre služby.

```
docker service create --name rabbitMqService rabbitmq:latest
```

Výpis 15: Vytvorenie Docker služby

V rámci služby je možné definovať počet inštancií, ktoré budú spustené parametrom –scale, prípadne parametrom replicas pre aktualizáciu. Ďalej vlastnosti, ktoré definujú čas medzi aktualizovaním jednotlivých inštancií či počet aktualizácií, ktoré budú bežať zároveň. Pre tvorbu clusteru je možné použiť Docker Swarm režim, ktoré predstavuje orchestračný nástroj pre existujúce inštancie. Predstavuje decentralizovaný návrhový vzor, podporuje škálovanie, správu siete, zabezpečenie, a správu dát či aktualizácií. V rámci klasteru môže fungovať manažérske a pracovné inštancie, ktoré neriadia ďalšie jednotky v klastri. Nástroj ponúka množstvo príkazov pre konfiguráciu. Alternativou orchestračného nástroju je Google Kubernetes, ktorý podporuje Docker kontajneri.

## 6.7 Sumarizácia implementácie

Výsledkom implementovaného riešenia sú 3 komponenty, ktoré využívajú pre prenos správ a ich ukladanie RabbitMq. Kapitola implementácie obsahuje popis pre jednotlivé z nich, kde boli zhrnuté prvky riešenia na úrovni zdrojového kódu, návrhu logiku pre spracovanie a konfiguráciu topológie spolu s návrhom postupu pre škálovanie riešenia formou služby či rýchleho otestovaniu využitím nástroju Docker.

Jednotlivé komponenty využívali databáze na úrovni kontajnerov a ich spustenie bolo prevádzané v testovacom prostredí s využitím hardvérových prostriedkov (Intel Core i5 8250U, 8GB RAM, SSD disk). Pre konkrétnu konfiguráciu boli spustené komponenty API rozhrania, koncového bodu a webového klienta pričom boli dosahované nasledovné výsledky. Pre záťažový test, ktorý simuloval backend – zasielanie požiadaviek na API rozhranie bol použitý nástroj Apache Ab a výsledky sú znázornené v tabuľke číslo 10.

Tabuľka 10: Výsledky testovania doručovania správ

| | |
|---|---|
| Počet požiadaviek - správ | 30 000 |
| Celkovo prenesených dát z API | 6870000 B |
| Veľkosť dát vo fronte | 4.1 MB |
| Veľkosť správ v pamäti procesu | 21 MB |
| Požiadavky za sekundu | 1446.45 |
| Priemerný čas požiadavky na API | 13.80 ms |
| Najvyšší počet doručovaných správ za sekundu | 750 |
| Priemerný počet doručovaných správ za sekundu | 450 |
| Najvyšší počet odoberaných správ za sekundu | 7500 |
| Priemerný počet odoberaných správ za sekundu | 4000 |
| Priemerná rýchlosť doručovania správy klientovi (API – klient) | 8.5ms |

*V – všetky verzie, tmavo vyznačené verzie nepodporujú web notifikácie*

- Počet spojení – v prípade častých opakovaných pripojení a vytváraní komunikačných kanálov môže dochádzať k vyššej záťaži.

- Veľkosť prenášaných správ. V prípade aplikácie aplikačnej logiky na komponente koncového bodu je nutné zvážiť dodatočné operácie či volania na externé zdroje.
- Počet vytvorených kanálov, front či spojení, ktoré môžu alokovať odhadom 100 Kb pamäte RAM, v prípade front približne 10 Kb a viac.
- Počet nepotvrdených správ, ktoré sú uložené celý čas v RAM.
- Vyšia réžia pri komunikácií využitím šifrovaného spojenia.

Prvky využité pri implementácií:

- Spoločné nakonfigurované smerovače, do ktorých zasiela API rozhranie.
- V prípade inzerátov, prenášanie identifikátoru, kedy sú samotné dáta spracované koncovým bodom. Výhodou je možnosť dodatočnej aplikačnéj logiky ako napríklad rôzny formát webovej notifikácie pre rôznych používateľov.
- Privátne smerovače a následne mapované fronty pre koncový bod. V prípade vysokého počtu front by mohli vznikať stovky mapovaní na jeden konkrétny smerovač. Vopred konfigurované smerovače preposielajú na privátne smerovače a následne je správa smerovaná do príslušnej fronty konkrétneho koncového bodu.
- Dočasné fronty s exspirácia neprebratých správ pre notifikácie, ktoré predstavujú informačný charakter.
- Asynchrónne spracovanie udalosti pre odoslanie notifikácie klientovi

### 6.7.1 Porovnanie s existujúcimi riešeniami

Typickým riešením, ktorým sa zaoberajú existujúce projekty je práve doručovanie notifikácií v reálnom čase. Tie môžu zahrňovať rôzne formáty dát, zdrojovú či cieľovú destináciu. Implementované riešenie zobrazuje na uvedenom príklade prehľad fungovania jednotlivých komponent a proces prenosu správ. Výhodou implementácie taktiež možnosť prispôsobenia na konkrétnu aplikačnú logiku vďaka dostupným rozhraniam, úprave či pridanú nových formátov notifikácií, aplikácií filtru správ a prípadnej úpravy topológie RabbitMq na základe uvedeného vzoru. Prípadným škálovaním vieme zabezpečiť dostatočný výkon aj v prípade niekoľko tisíc spojení s dosiahnutým komunikácie v reálnom čase.

Výhodou existujúcich riešení je možnosť prispôsobenia a využitia existujúcej infraštruktúry. V prípade služieb ako Google Firebase a iných je možnosť využívania voľných kvót pre prenos, ktoré často dostačujú pre menšie projekty. Medzi ďalšie prvky, ktoré sú v prípade využitia externého riešenia dostupné sú správa úložiska, analytika prenosu správ, zabezpečenie spojenia, podpora rôznych klientov a protokolov vrátane mobilných zariadení, spolupráca s Apple Push Notification Service či Firebase Messaging a na koniec celková správa infraštruktúry a jej

škálovanie. Implementované riešenie je tak na záver vhodne ako prípadne testovacie prostredie či komponenta pre menšie projekty, kde nie sú tak vysoké požiadavky na prípadne škálovanie služby, ktorej problematika často predstavuje náročnejšie riešenie ako samotná implementácie komponentov.

# 7 Získavanie dát medzi rôznymi typmi úložísk

Častým prvkom, ku ktorému dochádza počas vývoja aplikácií je optimalizácia či už z pohľadu využitia úložiska a jednotlivých lokálnych zdrojov, na ktorom sa nachádzajú dáta alebo z pohľadu rýchlosti správnosti výsledku. Jedným zo spôsobov ako optimalizovať získavania dát, ktoré majú vlastnosti ako dáta, ktoré sa po určitú dobu nemenia, vieme prípadne agregovať jednotlivé menšie časti do väčších štruktúr a jednoduchším prístupom ich naraz získať.

Pre existujúce databáze existujú operácie, ktoré v rámci vykonávania dotazov a iných operácií vytvárajú dočasné dáta, ktoré vedia znovu použiť v prípade opakovaného dotazu. Naopak pri technológiách akými sú napríklad Redis či Memcached vieme uchovávať dáta oveľa viac optimalizované hlavne čo sa týka neskoršieho prístupu. Pri nastavení pravidiel cachovania je podmienkou určiť, ktoré dáta sa budú do cache ukladať a kedy budú exspirovaná, teda lepšie povedané, odstránené z cache. Taktiež musíme brať do úvahy využitie systémových zdrojov, hlavne čo sa týka pamäte RAM, do ktorej sú najčastejšie dáta ukladané pre rýchli prístup a vyšší výkon oproti diskovým operáciám.

Jedným zo spôsobov akým využíva cache práve dokumentovo orientovaná databáza je ukladanie výsledkov na úrovni shardov, ktoré ukladajú dáta v podobe výsledkov jednotlivých dotazov. V prípade behu rovnakého dotazu, ktorý je v rámci cache identifikovaný pomocou reťazcu hash môžu byť časti dotazu vykonané rýchlejšie nakoľko sú znovu použité.

## 7.1 Redis

Z predošlej kapitoly porovnávania NoSQL databáz, kde sa nachádza aj Redis môžeme vidieť najlepšie výsledky najmä z hľadiská rýchlosti vyhľadávania záznamu. Uloženie dát je v pamäte a prístup k ním veľmi efektívny a preto sa v aktuálnom riešení hodí táto databáza na účel dátovej cache. Pre tieto účely obsahuje niekoľko zaujímavých vlastností. V prípade LRU cache môžeme definovať pravidlá, na základe ktorých budú v budúcnosti zmazané nevyužívané dáta prípadne náhodne za účelom uvoľnenia systémových prostriedkov, najmä pamäte RAM. Z hľadiska konfigurácie je možné nastaviť pamäťový limit, kedy v prípade jeho naplnenia dôjde k aplikovaniu politiky.
Tá môže byť konfigurovaná nasledovnými pravidlami:

- Chyba v prípade naplnenia cache.
- Odstránenie LRU dát.
- Odstránenie dát, ktoré majú nastavenú exspiráciu, prípadne sa ich exspirácie blíži ku koncu.
- Náhodné zmazanie dát, prípadne náhodne pre dáta, ktoré majú nastavenú exspiráciu.

Implementácia v rámci Redisu predstavuje aproximačný algoritmus pričom výber dát, ktoré vyradí zakladá na najstaršom čase prístupu. Konečný výber je zo skupiny kandidátov a tak si blíži k reálnemu LRU algoritmu. Od verzie 4.0 je dostupný LFU algoritmus, ktorý sleduje prístup k jednotlivým dáta a o vyradení rozhoduje na základe tohto údaju. Ku kontrole obsadenie pamäte dochádza pred vložením nového záznamu.

Pre jednotlivé ukladané kľúče je možné konfigurovať číselnú hodnotu exspirácie, ktorá vyjadruje čas ako dlho bude záznam uložený v pamäti. K overeniu exspirácie môže dôjsť pri snahe získať záznam prípadne počas aktívneho odstraňovania, kedy Redis približne 10 krát za sekundu prevádza testy exspirácie nad skupinou uložených dát využitím pravdepodobnostného algoritmu. V prípade aktualizácie záznamu je možné exspiráciu predĺžiť.

## 7.2 Elasticsearch

Vybraná dokumentovo orientovaná databáza boli popísaná a porovnaná v predošlých kapitolách. V prípade využitia pre cache môžeme chcieť využiť výsledky z rôznych dotazov, využiť uloženie na nezávislej databáze a dosiahnuť tak prípadne zvýšenie výkonu najmä vyhľadávania. V rámci Elasticsearch môžeme rozdeliť jednotlivé dotazy na skupiny dotazov, pri ktorých prevádzame filtrovanie, full-textové vyhľadávanie, agregované dotazy, pri ktorých môžu byť tiež aplikované filtre a v neposlednom rade geolokačné dotazy. Z hľadiska účelov cache môžeme chcieť uchovávať výsledky vyhľadávania, kedy nie je podmienkou mať pri každom požiadavku v určitom časovom rozmedzí aktuálne dáta. Príkladom môže byť vyhľadávanie produktov či údaje vo filtroch, ktoré obsahujú veľké množstvo dokumentov.

Z hľadiska práce s dočasnými dátami dokáže využívať Elasticsearch cache požiadaviek na úrovni cache. Výsledné dáta dokáže kombinovať z uložených dát jednotlivých shardov. Využitie je najmä pri náročných dotazoch na operácie vyhľadávania. Štandardne sú ukladané medzivýsledky v prípade dotazov, kde nie je definovaná veľkosť výsledku kedy ukladá dáta o celkovom počet dokumentov vo výsledku, agregácie a výsledky z nápovedy. Túto vlastnosť je možné zmeniť v nastaveniach indexu prípade využiť clear-cache API (_cache) a následne využiť URL boolean parameter "request". Štandardným nastavením veľkosť cache je 1% z celkového úložného priestoru. Oproti Redisu môžeme sledovať rôzne správanie pri automatickom cachovaní podobne využíva sledovanie využitia konkrétneho záznamu a odstraňuje implementovaným algoritmom LRU avšak výsledky, ktoré majú napríklad menej ako 10 000 záznamov nemusia byť uložené a vyhľadávanie využíva aj v prípade filtrácie invertovaný index, ktorý je rýchlejší ako operácie uloženia do cache. [36]

## 7.3 Implementacia rozhrania

Implementácie predstavuje aplikáciu vytvorenú v NodeJs podobne ako v predošlej kapitole. Pre účely dotazovania bolo vytvorené API rozhranie, s koncovými bodmi uvedenými v tabuľke 11, ktoré zaobaľuje volania, v konkrétnom prípade medzi databázou Redis a Elasticsearch. Uvedené

riešenie pracuje s dotazmi na získanie výsledkov počtu dokumentov a samotného vyhľadávacie dotazu, ktorý môže zahrňovať filtrovanie či agregačnú časť.

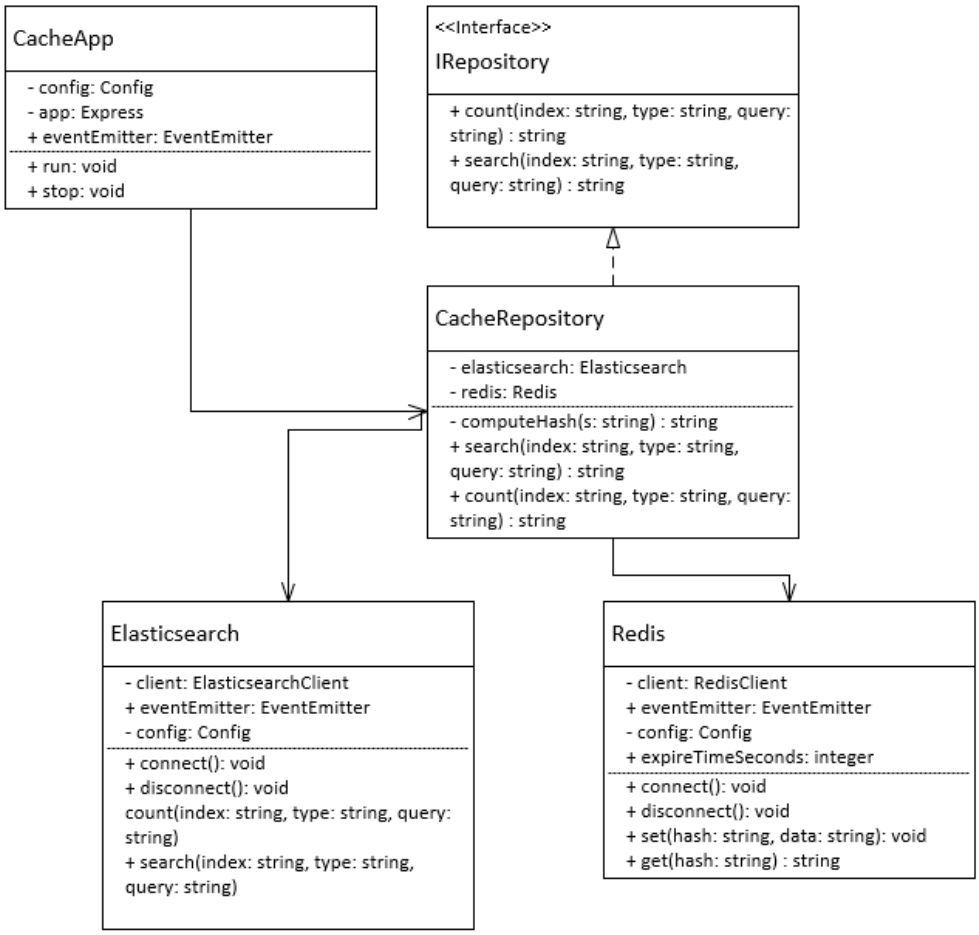
Tabuľka 11: Prehľad koncových bodov cache API rozhrania

| Koncový bod | Parametre | Metóda | Popis |
|---|---|---|---|
| /count/ | Index: string, type: string, query: string | POST | Získanie počtu výsledkov |
| /search/ | Index: string, type: string, query: string | POST | Získanie konkrétnych dokumentov |

Implementácie klientov Elasticsearch a Redis využívajú NPM knižnice pre NodeJs. Ako je uvedené na diagrame nižšie, predstavujú asociatívne spojenie s triedou CacheRepository, ktorá obsahuje logiku pre výpočet hash hodnoty a spracovanie dotazu. Výpočet hash hodnoty prebieha na základe vstupných parametrov, ktorá zahŕňajú názov dotazovaného indexu, typ dokumentu a telo dotazu. Pre výpočet je použitý algoritmus SHA-256, ktorý vychádza zo skupiny algoritmov SHA-2 a počíta využitím 32-bitových slov. Algoritmus u v riešenom prípade zaručuje vygenerovanie dostatočne unikátnej hodnoty kľúča pre konkrétnu hodnotu.

Pre uloženie hodnoty do Redisu je použitý príkaz SET, ktorý podporuje parameter exspirácie a je ho možné konfigurovať ako konfiguračný parameter. Časová komplexita pri vyhľadávaní dosahuje O(1). V prípade úspešného výsledku vráti Redis reťazec Ökä prázdny v prípade chyby. V prípade zaručenie jednoznačného prístupu je možné konfigurovať zámok na jednotlivými záznamami, ktorý obsahuje atribút názvu zdroju a čas ako dlho bude záznam uzamknutý.

Výsledkom je rozhranie na ukladanie výsledkov do cache medzi uvedenými databázami. V prípade rozšírenie pre ďalšie databáze je možné pridať nové koncové body a vytvoriť oddelenú triedu, ktorá bude predstavovať repozitár, ktorý komunikuje s úložiskom a samotná implementácia voči rozhraniu nie je závislá priamo na type danej databáze. (Schéma 20)




Obr. 20: UML triedny diagram implementovaného cache rozhrania

# 8 Záver

Cieľom diplomovej práce bolo implementovať systém pre správu notifikácií, ktorá obsahuje procesy od odoslania až po doručenie. Samotnej implementácií predchádzala analýza, ktorá bola zameraná na definovanie funkčných požiadaviek predstavujúcich webovú aplikáciu vo forme automobilového inzerentného portálu, ktoré popisujú akcie, pri ktorých by mohol byť vytvorený objekt notifikácie, ktorý je následne vo forme webovej notifikácie doručený konkrétnemu používateľovi či skupine. V časti návrhu boli využité poznatky z predošlých kapitol práce, ktoré viedli k vytvoreniu jednotlivých komponentov systému tak aby dokázali medzi sebou komunikovať bez priamej závislosti na implementácií ostatných častí. Súčasťou práce bola analýza spôsobov ukladania dát a dotazovania. Kapitola porovnania je zameraná na popis významnejších vlastností jednotlivých technológií spolu s doplnením informácií o možnostiach vertikálneho či horizontálneho škálovania pre účely zvýšenia výkonu či dosiahnutia lepšej stability. V ďalšej časti analýzy boli spracované informácie o existujúcich službách či protokoloch pre zasielanie push notifikácií, ktoré je možné doručovať medzi webových a mobilných klientov. Vytvorené API rozhranie ponúka jednoduchý spôsob pre vyhľadávanie dát medzi úložiskami Elasticsearch a Redis, ktorý zohráva úlohu pre dočasné uloženie dát vo forme cache. Kapitola porovnáva vlastnosti cache oboch technológií a ponúka možnosti rozšírenia pre ďalšie technológie. Hlavný bod práce, systém pre prenos správ, ktorý bol vopred spomenutý predstavuje okrem navrhnutého riešenia detailnejší pohľad na fungovanie podobnej architektúry, ktorú je vhodné využiť ako vzor pre integráciu rozličných systémov spolu s návrhom topológie pre konkrétnu technológiu, ktorá by prenášala a ukladala správy. Implementácia popisuje použité nástroje, postupy a vrstvy aplikácie a následný úvod do kontajnerovej technológie Docker pomocou, ktorej by bolo možné využívať jednotlivé komponenty vo forme služby a škálovať ju medzi viacero zariadení. Popísané poznatky tak môžu podať odpoveď pri rozhodovaní využitia vlastnej či externej služby.

# Literatúra


[1] Elasticsearch Reference [online]. [cit. 2018-10-03]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/5.6/index.html

[2] Apache Solr Reference Guide [online]. [cit. 2018-10-03]. Dostupné z: https://lucene.apache.org/solr/guide/7_7/

[3] Apache Cassandra Documentation v4.0 [online]. [cit. 2018-10-03]. Dostupné z: http://cassandra.apache.org/doc/latest/cql/index.html

[4] Apache HBase ™ Reference Guide [online]. [cit. 2018-10-03]. Dostupné z: http://cassandra.apache.org/doc/latest/

[5] MemSQL Documentation [online]. [cit. 2018-10-03]. Dostupné z: https://docs.memsql.com/

[6] The MongoDB 4.0 Manual [online]. [cit. 2018-10-03]. Dostupné z: https://docs.mongodb.com/manual/

[7] Documentation [online]. [cit. 2018-10-03]. Dostupné z: https://redis.io/documentation

[8] WILDER, Bill. Cloud architecture patterns. Tokyo: O'Reilly, [2012]. ISBN 14-493-1977-7.

[9] Sharding [online]. [cit. 2018-10-13]. Dostupné z: https://docs.mongodb.com/manual/sharding/

[10] Introduction to Scaling and Distribution [online]. [cit. 2018-10-13]. Dostupné z: https://lucene.apache.org/solr/guide/6_6/introduction-to-scaling-and-distribution.html

[11] DynamoDB vs. Cassandra [online]. [cit. 2018-10-13]. Dostupné z: https://www.kdnuggets.com/2018/08/dynamodb-vs-cassandra.html

[12] Optimizing Table Data Structures [online]. [cit. 2018-10-13]. Dostupné z: https://docs.memsql.com/tutorials/v6.5/optimizing-table-data-structures/

[13] How Scaling Really Works in Apache HBase [online]. [cit. 2018-10-13]. Dostupné z: https://blog.cloudera.com/blog/2013/04/how-scaling-really-works-in-apache-hbase/

[14] How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances [online]. [cit. 2018-10-13]. Dostupné z: http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html

[15] YAHIAOUI, Houssem. Firebase Cookbook. Birmingham: Packt Publishing, [2017]. ISBN 1788296338.

[16] Product Overview [online]. [cit. 2019-11-01]. Dostupné z: https://documentation.onesignal.com/docs

[17] Pub/Sub [online]. [cit. 2018-11-01]. Dostupné z: https://redis.io/topics/pubsub

[18] FOWLER, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, c2003. Addison-Wesley signature series. ISBN 03-211-2742-0.

[19] Introduction to Message Transformation [online]. [cit. 2019-02-15]. Dostupné z: https://www.enterpriseintegrationpatterns.com/patterns/messaging/

[20] Introduction to Message Routing [online]. [cit. 2019-02-15]. Dostupné z: https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRoutingIntro.html

[21] Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions 1st Edition. Kanada: Addison-Wesley Professional, [2003]. ISBN 9780321200686.

[22] NEWMAN, Sam. Building Microservices: Designing Fine-Grained Systems. Sebastopol: O'Reilly Media, [2015]. ISBN 1491950358.

[23] What is Push Notifications And How it Works? [online]. [cit. 2019-02-22]. Dostupné z: https://www.pushmaze.com/what-is-push-notifications/

[24] Can I use Web Notifications [online]. [cit. 2019-02-21]. Dostupné z: https://caniuse.com/#feat=push-api

[25] Web Notifications [online]. [cit. 2019-02-22]. Dostupné z: https://www.w3.org/TR/notifications/

[26] HTTP status and error codes for JSON [online]. [cit. 2019-03-11]. Dostupné z: https://cloud.google.com/storage/docs/json_api/v1/status-codes

[27] APNs Overview [online]. [cit. 2019-03-11]. Dostupné z: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html#//apple_ref/doc/uid/TP40008194-CH8-SW1

[28] TARKOMA, Sasu. Publish/subscribe systems: design and principles. Hoboken, N.J.: Wiley, 2012. ISBN 978-1119951544.

[29] Developer FAQ's [online]. [cit. 2019-03-25]. Dostupné z: https://www.amqp.org/resources/developer-faqs

[30] Core Documentation [online]. [cit. 2019-03-25]. http://zeromq.org/intro:read-the-manual

[31] Server Documentation [online]. [cit. 2019-03-25]. https://www.rabbitmq.com/admin-guide.html

[32] Documentation [online]. [cit. 2019-03-25]. Dostupné z: https://qpid.apache.org/documentation.html

[33] Documentation [online]. [cit.2019-03-25]. Dostupné z: https://kafka.apache.org/documentation/

[34] ActiveMQ 5 [online]. [cit. 2019-03-25]. Dostupné z: http://activemq.apache.org/components/classic/

[35] SQS Developer Guide [online]. [cit. 2019-03-25]. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html

[36] Shard Request Cache [online]. [cit. 2019-04-13]. https://www.elastic.co/guide/en/elasticsearch/reference/5.6/shard-request-cache.html

# A Dotazy

### Elasticsearch dotaz

```
"query": {
"bool": {
"must": { "match_all": {} },
"filter": {
"range": {
"balance": {
"gte": 20000,
"lte": 30000
"aggs": {
"group_by_state": {
"terms": {
"field": "state.keyword"
}
}
```

Výpis 16: Ukážka dotazu s filtrom v Elasticsearch

### MongoDB dotaz

```
db.banks.aggregate(
[
{ $group: { "_id": "$accountField", "count": { $sum: 1 } } }
]
);
```

Výpis 17: Ukážka agregovaného dotazu v MongoDB

### Ukážka vytvorenia indexu s mappingom v Elasticsearch 5.6

```
PUT /nazov_indexu
{
"settings" : {
"number_of_shards" : 1
},
"mytype" : {
"properties" : {
```

```
"myproperty" : { "type" : "text" }
}
}
}
```

Výpis 18: Ukážka vytvorenia indexu v Elastisearch

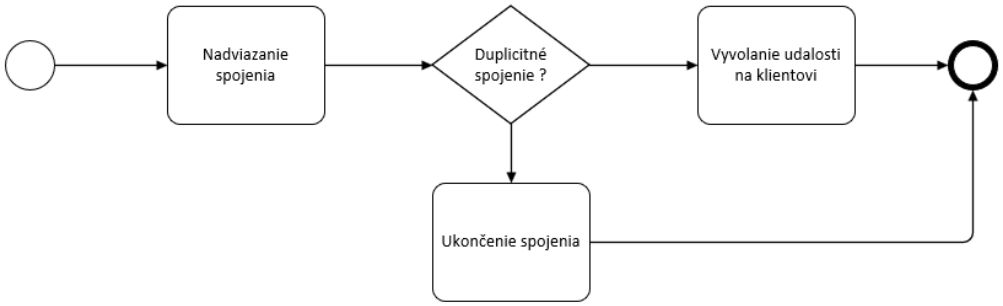
### Ukážka vloženia záznamu do Elasticsearch

```
PUT /nazov_indexu/typ/volitelne_id
{
"myproperty": "Diplomova praca"
}
```

Výpis 19: Ukážka vloženia záznamu do Elasticsearch

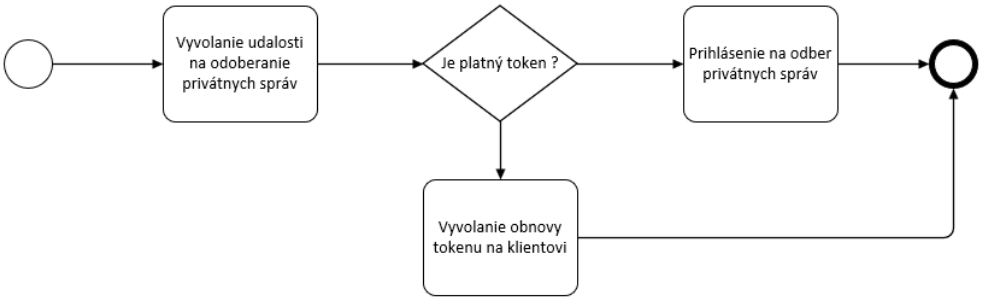
# B Diagramy

### BPNM diagram pre koncový bod – nadviazanie spojenia



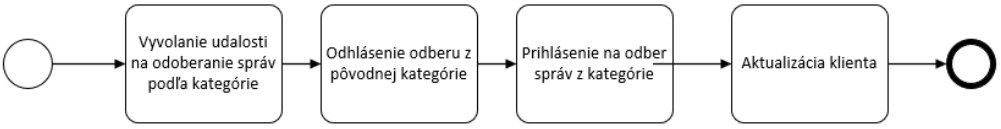

Obr. 21: Diagram pre koncový bod

### BPNM diagram pre koncový bod – prihlásenie na odber používateľských správ



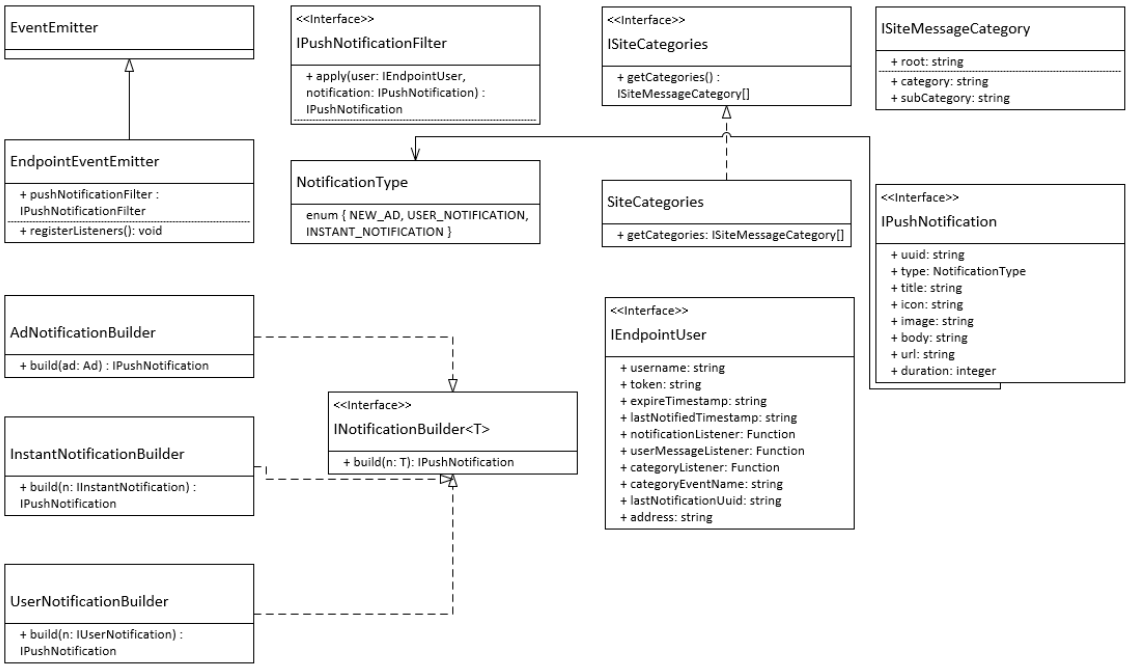

Obr. 22: Diagram pre prihlásenie používateľa

### BPNM diagram pre koncový bod – prihlásenie na správ podľa kategórie




Obr. 23: Diagram pre prihlásenie do kategórie

### UML triedny diagram dôležitých rozhraní a tried koncového bodu




Obr. 24: UML diagram koncového bodu

# Zoznam príloh

Súčasťou diplomovéj práce je priložené CD
Štruktúra CD obsahuje nasledujúce prílohy:

| Priečinok | Popis |
|---|---|
| Implementácia | Zdrojového kódy |
| Diagramy | UML diagramy komponentov |